Fachbereich III Informations- und Kommunikationswissenschaften

Institut für Angewandte Sprachwissenschaft

MAGISTERARBEIT

INTERNATIONALES INFORMATIONSMANAGEMENT

Objektorientierte Softwareentwicklung in virtuellen Teams

–

Modellierung und Ansätze zur automatischen Erkennung von Problemsituationen

vorgelegt von

Kerstin Bischoff

Erstgutachter: Prof. Dr. Christa Womser-Hacker

Zweitgutachter: Dr. Thomas Mandl

Hildesheim, im Oktober 2005

ZUSAMMENFASSUNG

Softwareentwicklung findet in der professionellen Praxis in der Regel in Teams statt. Da sich kollaborative Lernszenarien darüber hinaus als förderlich für den Wissenserwerb gezeigt haben, wird Programmieren in der universitären Lehre oft bereits im Team – in kooperativen Praxisteilen wie Gruppenübungen oder Softwarepraktika – gelernt. Neue Informations- und Kommunikationstechnologien ermöglichen es, diese Form des gemeinsamen Wissenserwerbs auch virtuell über zeitliche und geographische Grenzen hinweg durchzuführen. Ziel des Projektes VitaminL ist es, virtuelle Lerngruppen bei der Bearbeitung von Programmieraufgaben in einer netzbasierten Lernumgebung in Problemsituationen durch eine Softwarekomponente effektiv zu unterstützen.
In der vorliegenden Magisterarbeit werden zunächst die Anforderungen an eine derartige automatisierte fachliche Unterstützung ermittelt. Vor dem Hintergrund der (kognitionspsychologischen) Forschungsliteratur zur komplexen menschlichen Fähigkeit des Programmierens und speziell zu Anfängerproblemen werden mittels einer Analyse von virtuellen Programmiersitzungen (in der VitaminL-Plattform) typische Problemsituationen identifiziert. Anhand dieser empirischen Ergebnisse aus Beobachtungen und Tests werden verschiedene Unterstützungsmöglichkeiten diskutiert und die Realisierung eines beispielbasierten Ansatzes mittels maschineller Lernverfahren skizziert. Eine erste Evaluation auf der Grundlage einer Systemsimulation in Benutzertests (Wizard-of-Oz) zeigt die grundsätzliche Eignung des Ansatzes und die experimentelle Prüfung unterschiedlicher Lernverfahren dessen Umsetzbarkeit. In einer kritischen Reflexion werden Einschränkungen ebenso wie sinnvolle Erweiterungsmöglichkeiten aufgezeigt.

Schlagwörter: CSCL, Softwareentwicklung, Java, Programmieranfänger, automatisches Debuggen, ITS, Maschinelles Lernen, beispielbasiertes Programmieren, Logfile-Analyse

ABSTRACT

The complex process of software engineering is typically performed by teams. Moreover, since collaborative learning scenarios have proven to subserve knowledge acquisition, learning programming in groups is increasingly common in Higher Education. New information and communication technologies provide the opportunity to learn collaboratively across time and distance within so-called virtual teams. By means of developing a software agent (an automatic tutor), the VitaminL-Project aims at effectively supporting virtual teams of novice JAVA programmers while working and learning via a CSCL-platform. This thesis investigates typical problems of novice programmers taking into account relevant literature from cognitive science/psychology. Based on empirical results from observations and tests within the framework of VitaminL different techniques for program analysis and tutorial support are discussed. Finally, an example-based approach is followed for which a prototypical implementation employing Machine Learning is outlined. A Proof-of-concept evaluation using a Hidden-Operator-Simulation (or Wizard-of-Oz) shows the usefulness of examples as well as the feasibility of the Machine Learning approach. Critically reflecting upon this work, an outlook on and recommendations for future work like a detailed (technical) evaluation and beneficial extensions round off this thesis.

Keywords: CSCL, Software engineering, Java, novices, programming, automated debugging, ITS, Machine Learning, example based programming, log file-analysis

EINLEITUNG

Softwareentwicklung ist ein komplexer Prozess, dessen Produkt in der Praxis oft nur unter Aufwendung enormer Ressourcen sowohl materieller als auch personeller Art bewerkstelligt werden kann. In großen Teams müssen Anforderungen ermittelt, Systeme entworfen, die einzelnen Module implementiert und ausgiebig getestet werden sowie die Wartung und Weiterverwendbarkeit gesichert werden.

Um auf diese Teamarbeit sowohl fachlich als auch in Bezug auf die erforderlichen Metakompetenzen vorzubereiten, vor allem aber, da sich kollaborative Lernszenarien als förderlich für den Wissenserwerb gezeigt haben [vgl. Slavin 1990; vgl. Pfister & Wessner 2001, 252f], wird Programmieren oft bereits im Team gelernt. So werden in der universitären Lehre Einführungskurse in die Programmierung zumeist durch kooperative Praxisteile wie Gruppenübungen oder Softwareprojekte ergänzt. Gleichzeitig ermöglichen es die neuen Informations- und Kommunikationstechnologien diese Form des gemeinsamen Wissenserwerbs auch virtuell über zeitliche und geographische Grenzen hinweg durchzuführen. Im Blickfeld des Forschungsfeldes CSCL (Computer-Supported Collaborative/ Cooperative Learning) steht dabei, wie die Gruppen in ihrem Lernprozess durch geeignete Lernumgebungen und Werkzeuge in ihrer Kommunikation und Koordination unterstützt werden können [vgl. Pfister & Wessner 2001, 251-256]. Das Erlernen einer Programmiersprache stellt für Anfänger zumeist eine anspruchsvolle Herausforderung dar und Tutoren nehmen in problemorientierten Übungen eine zentrale Stellung ein. Daher ist bei einer Übertragung in eine netzbasierte Lernumgebung der Aspekt der fachlichen Hilfestellung besonders zu beachten.

Darauf basiert das Anliegen des Projektes VitaminL (*Vi*rtuelle *T*eams: *A*nalyse und *M*odellierung *i*n *n*etzbasierten *L*ernumgebungen), virtuelle Lerngruppen bei der Bearbeitung von Programmieraufgaben über eine CSCL-Umgebung in Problemsituationen durch eine Softwarekomponente effektiv zu unterstützen [Kölle & Langemeier 2004a, b]. Um die Anforderungen an eine derartige automatisierte Unterstützung festlegen zu können, werden in der vorliegenden Arbeit typische Problemsituationen von Anfängern beim verteilten kollaborativen Lernen der Programmiersprache Java untersucht und ein Ansatz zur deren Überwindung vorgeschlagen. Für die Analyse wurden dazu Daten aus Benutzertests sowie aus dem ersten reellen Einsatz in einer virtuellen Kooperationsveranstaltung der Universitäten Hildesheim und Konstanz zur Einführung in die Programmiersprache Java gesammelt.

Zur Einordnung des Projektes VitaminL wird zuerst ein kurzer Überblick über das Forschungsgebiet CSCL gegeben und anschließend das Projekt ausführlicher vorgestellt. Aufgrund der Zielsetzung von VitaminL, die Unterstützung von Programmieranfängern, werden im nachfolgenden Kapitel Ergebnisse psychologischer Untersuchungen zu den kognitiven Prozessen beim Programmieren und typischen Fehlern während des Lernens einer Programmiersprache zusammengefasst. Um den Anfängern bei der Erkennung und Korrektur solcher Fehler Hilfestellung leisten zu können, sind verschiedene Techniken und Systeme entwickelt worden, deren Darstellung in Kapitel 3 erfolgt. Da in den Studien zu Anfängerschwierigkeiten und auch bei der Umsetzung intelligenter Systeme zu deren Überwindung zumeist individuell Lernende im Mittelpunkt stehen, werden im Kapitel 4 sodann Befunde aus der empirischen Beobachtung von Anfängern beim Zusammenarbeiten in VitaminL beschrieben.

Die Analyse erfasst dabei nur fachliche Problemsituationen, nicht aber solche, welche die Entwurfsgestaltung betreffen. Diese Fragestellung greift hingegen Göldner [2005] auf. Anhand der Resultate lassen sich die vorgestellten Unterstützungsmöglichkeiten hinsichtlich ihres Mehrwerts für eine Integration in die VitaminL-Plattform differenziert betrachten. So wird in Kapitel 5 anschließend an diese Diskussion der Ansatz der beispielbasierten Programmierung aufgegriffen und eine prototypische Realisierung mittels maschineller Lernverfahren skizziert. Der qualitativen Evaluation des Ansatzes auf der Grundlage von Benutzertests und experimenteller Prüfung verschiedener Lernverfahren folgt als Abschluss eine kritische Reflexion der Ergebnisse.

1 CSCL

CSCL (*C*omputer *S*upported *C*ooperative/*C*ollaborative[1] *L*earning) ist ein relativ junges Forschungsfeld, das in Abgrenzung zur langen Tradition individueller computerunterstützter Lehransätze die Bedeutung kooperativer Lernprozesse betont und Möglichkeiten zu deren Unterstützung durch neue Medien untersucht [vgl. Pfister & Wessner 2001, 251]. Nach Pfister & Wessner [2001, 251] kann CSCL

> „als eine Lernform definiert werden, in der mehrere Personen (mindestens zwei) unter (nicht unbedingt ausschließlicher) Nutzung von Computern ein Lernziel verfolgen, indem sie über den Lehrinhalt kommunizieren und neues Wissen kooperativ aufbauen. Kooperativ bedeutet, dass die Erreichung des Lernziels ein von allen Beteiligten geteiltes Ziel darstellt."

Kooperativen Lernformen liegt zumeist eine Auffassung des menschlichen Lernprozesses zugrunde, wie sie kognitivistische und pädagogisch-konstruktivistische Ansätze vertreten [vgl. Pfister & Wessner 2001, 252f]. Die zentralen Annahmen dieser Lerntheorien beschreiben Lernen als einen individuellen und selbstgesteuerten Konstruktionsprozess, der durch die soziale Umwelt angeregt wird und erst „im sozialen Disput durch die Bewertung von Informationen und den Austausch … mit anderen" [Zumbach 2003, 36] zu tiefer gehendem Verstehen und nachhaltigem Wissen führt [vgl. Zumbach 2003, 35f; Siebert 2003, 16-22]. Da Lernen zudem stets kontextgebunden stattfindet, ist die „Einbettung in realistische und relevante Kontexte" [Zumbach 2003, 37] z. B. durch Bearbeitung möglichst authentischer Probleme ein entscheidender Faktor für die Nachhaltigkeit und Anwendbarkeit des Wissens [vgl. Zumbach 2003, 36f; Pfister & Wessner 2001, 252ff].

Dementsprechend sollen Lernumgebungen geschaffen werden, die ein selbstgesteuertes und eigenverantwortliches Lernen durch Eigenkonstruktion anregen sowie metakognitive Reflexion bspw. durch Auseinandersetzung mit anderen fördern. Sie sollen Lernen in einen sozialen und praxisrelevanten Kontext stellen und gleichzeitig Lernen unter multiplen Perspektiven ermöglichen [vgl. Zumbach 2003, 37; Pfister & Wessner 2001, 252f]. Kooperatives Lernen hat sich praktisch oft als effektiver als individuelles Lernen gezeigt [vgl. Pfister & Wessner 2001, 253], obwohl negative Phänomene wie Trittbrettfahrereffekte und als Reaktion darauf Frustration bei den 'Arbeitstieren' oder das Sichzurückziehen einzelner Gruppenmitglieder bekanntlich immer wieder auftreten [vgl. Mayrberger 2004, 45f]. Neben aktiver Wissenskonstruktion und Wissenstransfer inner-

[1] Die Bezeichnungen variieren. Im deutschen Sprachgebrauch werden zumeist beide Begriffe gleichwertig benutzt, ohne im Grad der Zusammenarbeit zu unterscheiden [vgl. Mayrberger 2004, 39ff]

halb der Gruppe weist Ben-Ari [1998, 261] der Gruppe für das Lernen einer Programmiersprache auch eine wichtige motivationale Bedeutung zu:

> „Group assignments and closed labs should be preferable to individual homework exercises, because they soften the brutality of the interaction with the computer and facilitate the social interaction that is apparently necessary for successful construction."

Geringere Frustration, reduzierte Abhängigkeit vom Tutor in Problemsituationen und gesteigerte Lernerfolge berichten auch u. a. Williams et al. [2002], die in Anlehnung an Techniken des 'Extreme Programming' die strukturierte Form des 'Pair Programming' in einem Java-Einführungskurs untersucht haben. In Zweierteams wird an einem Computer programmiert, wobei der Tippende die Rolle des 'drivers' mit Design und Implementierung des Codes einnimmt und der andere Partner als 'navigator' auf Defekte prüft und strategisch plant [vgl. Williams et al. 2002, 197f]. Durch das regelmäßige Tauschen der Rollen wird dabei versucht, eine ausgeglichene Involvierung beider Partner und somit einen vergleichbaren Wissensstand zu sichern [vgl. Williams et al. 2002, 199ff].

CSCL-Systeme verfolgen das Ziel, computergestützte Lernumgebungen so zu gestalten, „dass hierbei die spezifischen Vorteile 'Kooperativen Lernens' aus nicht-computerunterstützten Lernsituationen zum Tragen kommen" [Mayrberger 2004, 47]. Im Gegensatz zum CSCW (Computer Supported Cooperative/Collaborative Work), dem Ursprung der wissenschaftlichen Disziplin des computerunterstützten kooperativen Lernens, zeichnen sie sich folglich durch ihre pädagogisch-didaktische Komponente mit Betonung auf dem gemeinsamen Wissenserwerb aus [vgl. Pfister & Wessner 2001, 251]. Bei der Umsetzung dieser generellen Zielformulierung lassen sich CSCL-Systeme entlang verschiedener Dimensionen wie Ort, Zeit, Symmetrie des Wissenstransfers, Grad der Selbststeuerung, Dauer der Zusammenarbeit, Verteilung des Wissens und der Gruppengröße einteilen.

Zudem variieren sie hinsichtlich der zur Verfügung gestellten Kommunikationskanäle und Kooperationswerkzeuge [vgl. Pfister & Wessner 2001, 251ff]. Ein Mehrwert kann so grundsätzlich durch die Integration verschiedener medialer Repräsentationsformen, komplexer Simulationen und Werkzeuge für das Explizieren von Wissen – z. B. durch strukturierte Diskurse – sowie das Speichern dieser expliziten Repräsentationen geschaffen werden [vgl. Pfister & Wessner 2001, 255].

Potenziale bieten besonders verteilte internetbasierte Systeme, in denen bspw. große Gruppen „asynchron per E-Mail und Bulletin Board über längere Zeit einen Lerndiskurs

führen“[2] [Pfister & Wessner 2001, 254] oder kleine Gruppen „selbstorganisiert und synchron Wissen austauschen“ [Pfister & Wessner 2001, 255]. Gruppen, die auf diese Weise unabhängig von Ort und Zeit zusammenarbeiten, werden als virtuelle Teams bezeichnet [vgl. u. a. Mayrberger 2004, 48]. Neben der gewonnenen Flexibilität entstehen durch die Speicherbarkeit der Kommunikation und Kooperation über die Plattform zusätzlich vielfältige Analyse- und Unterstützungsmöglichkeiten.

Da das Fehlen der sozialen Präsenz einer Face-to-Face-Situation mit ihren impliziten Hinweisreizen die Koordination und Kommunikation virtueller Teams erschwert [vgl. Pfister & Wessner 2001, 256f], ist Unterstützung gerade für sie wichtig [vgl. Brusilovsky & Peylo 2003, 161]. Brusilovsky & Peylo [2003, 162] unterscheiden dabei drei prominente Ansätze: „adaptive group formation and peer help, adaptive collaboration support and virtual students”. Im erstgenannten Fall werden basierend auf Wissen über eine zu erfüllende Aufgabe und die Kenntnisse der einzelnen Personen aus dem 'Student Model', wenn förderlich, Teams zusammengestellt [u. a. Go et al. 1997] oder der geeignete Ansprechpartner für ein konkretes Problem ermittelt [u. a. Bishop et al. 1998]. Beim 'adaptive collaboration support' werden den Teams Informationen über ihre Kooperation z. B. das Partizipationsverhalten einzelner gespiegelt. Hingegen wird im letzten der drei Ansätze in Form eines Coaches oder simulierten Teammitglied eingegriffen, falls die Analyse der Kommunikation und Kooperation starke Abweichungen zu 'idealen' Kooperationssituationen aufweist oder Teamfunktionen nicht besetzt sind [vgl. Jermann et al. 2001, 326-329; Brusilovsky & Peylo 2003, 162].

Du Boulay & Vizcaino [2002, auch Contreras et al. 2000] bspw. simulieren in ihrer verteilten kollaborativen Programmierumgebung **HabiPro** einen virtuellen Studenten, der Passivität einzelner Mitglieder, 'off-topic' Konversationen oder fachliche Schwierigkeiten im Problemlöseprozess erkennt. Eine konkrete Situation wird zuerst anhand der Informationen aus den individuellen 'Student Models', einem Gruppenmodell (Informationen über Kenntnisse aus Übungen, über Fehler) und dem Modell über die möglichen Verhaltensweisen des simulierten Studenten klassifiziert. Die sich daraus ergebenden Strategien umfassen: Motivieren bestimmter Mitglieder zur Partizipation, Fokussieren der Aufmerksamkeit z. B. durch gezielte Fragen, Anpassen des Aufgaben-

[2] Nach Bauknecht [2000] bezeichnet ein Bulletin Board einen gemeinsamen Informationsraum, in dem Meldungen verschiedener Autoren thematisch zusammengestellt und einer anonymen Leserschaft zur Verfügung gestellt werden. Neben der Möglichkeit des selektiven Abonnements von News Groups zu einem Thema verfügt der Benutzer in der Regel über Werkzeuge zur Selektion, Betrachtung und zum Editieren von Themen und Beiträgen.

niveaus oder fachliche Hilfe in Form von Hinweisen oder ähnlichen Aufgaben mit Lösungen.

Auch im Projekt **VitaminL** [Kölle & Langemeier 2004a, b] wird eine verteilte CSCL-Umgebung entwickelt, die virtuelle Programmierteams durch Simulation eines Teammitglieds zielgerichtet unterstützen soll. Im Gegensatz zu HabiPro orientiert sich die Lernform dabei stärker am problembasierten Lernen, indem objektorientierte Programme durch die Kleingruppen in vollständiger Eigenarbeit zu erstellen sind. Vor allem aber liegt der Analyse der Kooperation in VitaminL ein bestimmtes Rollenmodell zugrunde, im Rahmen dessen die Hilfestellung erfolgen soll.

Grundsätzliche Annahme ist dabei, dass die personelle Zusammensetzung eines Teams mit den von seinen Mitgliedern eingenommenen Rollen und entsprechenden Funktionen ein kritischer Faktor für eine erfolgreiche Zielerreichung ist [Kölle & Langemeier 2004a, 3]. Während der synchronen, verteilten Zusammenarbeit über die Lernplattform, einer vollständig in Java entwickelten Client-Server-Applikation, werden daher die Teams anhand ihrer Kommunikation hinsichtlich ihrer Rollenstruktur analysiert, um fehlende Rollen später automatisch simulieren zu können.

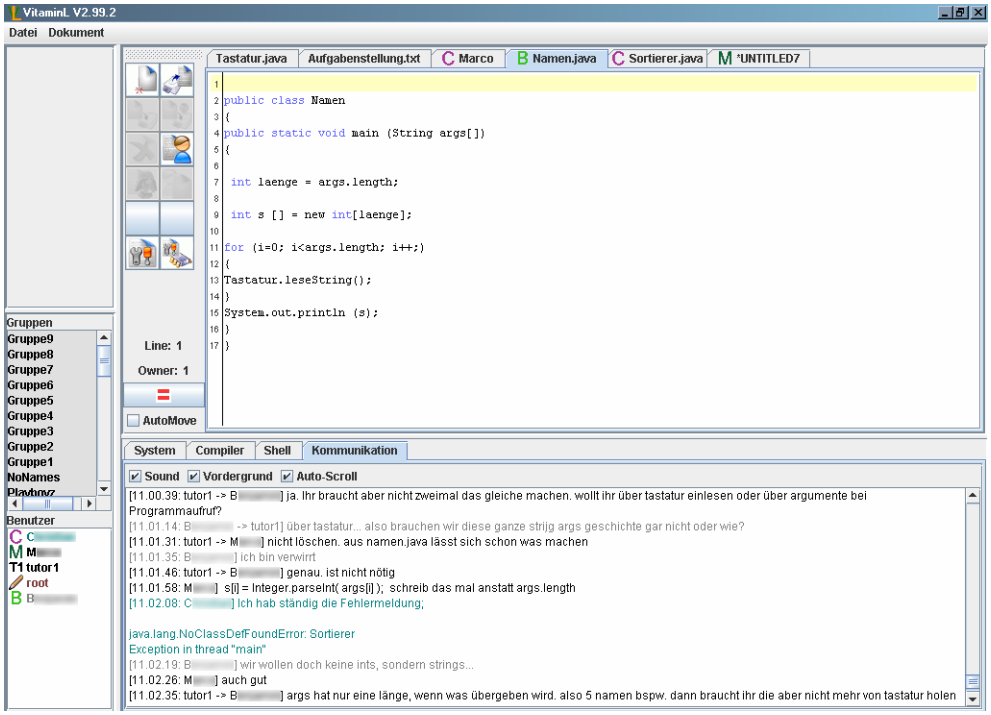

Abbildung 1. Die VitaminL-Umgebung

Grundlage für die Ermittlung dieser Rollen ist ein für den Anwendungsfall adaptiertes Rollenmodell, das die Persönlichkeitstypen in der objektorientierten Softwareentwicklung abbilden soll. In der praktischen Umsetzung erfolgt die Rollenzuweisung mittels

Persönlichkeitsfragebögen und einer strukturierten Kommunikationsschnittstelle innerhalb der VitaminL-Plattform. Teams kommunizieren während der gemeinsamen Programmierung über ein Chatfenster, in dem gemäß der 'Collaborative Learning Skills' von McManus & Aiken [1995; vgl. Kölle & Langemeier 2004a, 8] Kategorien für kommunikative Akte mit den zugehörigen typischen Satzanfängen ausgewählt und vervollständigt werden können.

Da die Analyse der Teamzusammensetzung auf der Zuordnung dieser Kommunikationsakte zu bestimmten Persönlichkeitstypen basiert, ist diese strukturierte Form der Kommunikation für das Ziel der effektiven Unterstützung durch Simulation fehlender Rollen eine wesentliche Grundvoraussetzung des Ansatzes. Alternativ besteht die Möglichkeit eines Freichats. Kernstück der Lernumgebung ist der Programmierbereich, in dem via 'Document Sharing' Java-Dokumente erzeugt, bearbeitet, untereinander ausgetauscht, kompiliert und ausgeführt werden können. Dabei kann jedes Teammitglied die Arbeit der anderen mitverfolgen. Alle lokalen Aktionen werden von den jeweiligen Clients an den Server und von dort an die übrigen Teammitglieder übermittelt und zum Zweck der Analyse und Evaluation in Form von Logfiles protokolliert[3] [Kölle & Langemeier 2004a, 11f].

Da in dieser Analyse in erster Linie fachliche – nicht kommunikative – Problemsituationen modelliert werden sollen, folgt nun ein Überblick über die Kenntnisse und Fertigkeiten, die es beim Programmieren zu erwerben gilt, sowie über empirische Ergebnisse zu charakteristischen Anfängerfehlern und -schwierigkeiten.

[3] Die Teilnehmer der Kooperationsveranstaltung wurden über das Loggen ihrer Sitzungen informiert und stimmten dem zu. Gleiches gilt für die freiwilligen Benutzertests. Zudem wurde für die Auswertung eine Anonymisierung vorgenommen.

2 PSYCHOLOGIE DES PROGRAMMIERENS

Programmieren ist eine komplexe Tätigkeit, an der verschiedene kognitive Prozesse beteiligt sind. Das Interesse an der Erforschung dieser Prozesse hat nach anfänglicher methodischer Kritik an experimentellen Designs und Repräsentativität [vgl. Sheil 1981] besonders in den 1980er Jahren einen umfangreichen Bestand an wissenschaftlichen Studien hervorgebracht. Zwei Richtungen sind dabei zu unterscheiden: Die Perspektive des Software-Engineering mit Betrachtung professioneller Programmierer, die zumeist in Teams große Projekte realisieren, und die pädagogisch-psychologische Erforschung des Programmierens als komplexer menschlicher Fähigkeit [vgl. Robins et al. 2003, 138ff]. Ähnlich dem umfassenden Überblick von Robins et al. [2003] soll hier die zweite Perspektive eingenommen werden und Wissenserwerb und Entwicklung der Fertigkeiten bei Programmieranfängern[4] im Mittelpunkt stehen. Als Grundlage für die Analyse typischer Anfängerfehler und -schwierigkeiten gibt der nächste Abschnitt eine kurze Einführung in das Programmieren aus kognitionspsychologischer Sicht. Dabei wird zuerst aufgezeigt, was einen Programmierexperten auszeichnet, um anschließend kontrastiv die Herausforderungen im Lernprozess und charakteristische Fehlerquellen sowie Problemfelder zu erörtern.

2.1 PROGRAMMIEREXPERTEN

Die Entwicklung von Programmen lässt sich zerlegen in verschiedene, stark miteinander interagierende Teilaufgaben, welche jeweils ihre eigenen Kenntnisse und Strategien erfordern. Programmieren umfasst damit die Analyse einer gegebenen Problemstellung, die Übertragung der zu erfüllenden Lösungsschritte in vom Computer berechenbare Algorithmen, die Implementierung des resultierenden Programmentwurfs und schließlich das systematische Testen und Beseitigen von Fehlern [vgl. Grabowski & Pennington 1990, 45f]. Hierbei handelt es sich um einen Problemlöseprozess, in dem durch Anwendung von Operationen[5] ein Ausgangszustand, die Anforderungen an ein zu erstellendes Programm, in einen Zielzustand, das entsprechende Programm, überführt wird. Von einem Problem wird in der Regel dann gesprochen, wenn für die Lösung

[4] Unter Anfängern werden hier Programmieranfänger in schulischer und universitärer Ausbildung verstanden, die weder bereits Erfahrung in einer anderen Programmiersprache besitzen noch über theoretische Grundlagen der Informatik verfügen.

[5] Die Operation sind die Teilschritte zur sukzessiven Erreichung des Zielzustands, während ein Operator die Handlung der Überführung insgesamt bezeichnet. [vgl. Edelmann 1996, 317]

nötiges Wissen nicht vorhanden ist, so dass für eine Aufgabe Einzelschritte und/ oder ihre Abfolge unbekannt sind [Zimbardo 1995, 375ff]. Generell zeichnen sich Experten eines Bereichs dadurch aus, dass sie über ein umfangreiches Repertoire an Operationen wie Regeln und Schemata sowie über effektive und effiziente Strategien und Heuristiken zur ihrer Auswahl und Kombination verfügen [Zimbardo 1995, 383]. Während Anfänger nötige Teilschritte aus ihrem soeben erworbenen deklarativen Faktenwissen konstruieren müssen, besitzen Experten ein durch die ständige Praxis fundiertes automatisiertes Handlungswissen und 'erinnern' diese Operationen [vgl. Anderson 2000, 282, 290]. Diese prozeduralen Fähigkeiten ermöglichen zum einen eine schnellere und fehlerfreiere Ausführung von Handlungen, zum anderen werden durch die Automatisierung kognitive Ressourcen für übergeordnete Aktivitäten wie strategisches Planen [vgl. Anderson 2000, 301] frei. Überdies unterscheiden sich Experten von Anfängern durch ihre metakognitiven Kompetenzen hinsichtlich der Einschätzung ihres Wissens und dessen Anwendbarkeit. Ferner ist ihr Wissen aufgrund der in multiple Kontexte eingebetteten Erfahrungen flexibler abrufbar [vgl. Zimbardo 1995, 383; Anderson 2000, 297f].

Für die Domäne des Programmierens schätzt man die notwendige Zeit, um von einem Anfänger über verschiedene Entwicklungsstufen zu einem Experten zu werden, auf etwa zehn Jahre [vgl. Winslow 1996, 18]. „(E)xpertise only comes with the investment of a great deal of time to learn the patterns, the problem-solving rules, and the appropriate problem-solving organization for a domain" [Anderson 2000, 303]. Das erforderliche Training setzt jedoch einen aktiven und reflektierenden Lernprozess voraus, in dem Feedback gesucht und genutzt wird, um das eigene Wissen zu vertiefen, zu erweitern und angemessen anzuwenden [Anderson 2000, 304].

Studien über Programmierexperten heben folglich auf die differenzierten Wissensrepräsentationen und Problemlösestrategien von Experten ab [vgl. Robins et al. 2003, 139]. Viele Untersuchungen haben so gezeigt, dass sie reichhaltige Strategien zur Problemrepräsentation und -umformulierung anwenden, was ihnen ermöglicht, Problemstellungen anhand der einzusetzenden Algorithmen zu klassifizieren [vgl. Winslow 1996, 18; Grabowski & Pennington 1990, 48f]. Da Programme in der Regel zur Lösung von Aufgaben aus anderen Domänen – Statistik, Finanzen, etc. – erstellt werden, unterscheiden sich diese auf den ersten Blick erheblich voneinander. Trotz oberflächlicher Unterschiede erkennen Experten jedoch schnell, welche Problemstellungen eine ähnliche Lösung verlangen. In Folge unzureichender Kenntnisse über das jeweilige Fach-

gebiet übersehen sie aber auch wichtige Anforderungen [vgl. Grabowski & Pennington 1990, 48f], was zu falschen oder unvollständigen Spezifikationen in der Planungsphase und folglich einem fehlerhaften Programm führen kann. Bei der Programmplanung zerlegen Experten der klassischen Ansicht nach in schrittweiser Verfeinerung 'top-down'-'breadth-first' das Problem sukzessive in eine Hierarchie von Unterzielen, wodurch sie Abhängigkeiten oder negative Wechselwirkungen zwischen den ermittelten Teilzielen schnell erkennen [vgl. Anderson 2000, 296; Grabowski & Pennington 1990, 50]. Neuere Ergebnisse beschreiben die Planung jedoch eher als einen explorativen Prozess, in dem zwischen schrittweiser Problemdekomposition und 'bottom-up'-Strategien ausgehend von bestehenden Code-Fragmenten opportunistisch gewechselt wird [vgl. Green 1990b, 117; Grabowski & Pennington 1990, 50f]. Als nötige Problemlöseoperationen verfügen Experten über detaillierte programmierspezifische Produktionsregeln, also Schemata bzw. Pläne[6] zur Erreichung bestimmter Teilziele in einem Programm.

> „A production rule consists of two propositions forming a 'condition-action' pair, one of which is a goal …, the other being the action or subgoals required to achieve that state. (…) If the conditions of a production rule are satisfied by matching them with a perceptual input, retrieval of knowledge from a long-term store, or by the action of another rule succeeding , then the rule is fired or 'activated' (i.e. made accessible to conscious attention) and its action is undertaken" [Ormerod 1990, 67].

Während Produktionsregeln eine erfolgreiche frühere Anwendung von Operationen zur Lösung eines bestimmten Ziels als kompilierte Strategien enkodieren und damit eher dem automatisierten prozeduralen Wissen zugeordnet werden, betonen Schemata bzw. Pläne die Form der Wissensrepräsentation des deklarativen Faktenwissens [vgl. Anderson 2000, 250ff; Ormerod 1990, 67ff].

> "The study of human problem solving shows that template-like solutions, or Plans, are used in solving a problem. In programming, a Plan consists of related pieces of code representing a specific action. The Plan is implemented using Language Constructs and can be as short as a single statement … or as long as an entire group of codes which may itself contain other plans, e.g. Sort Plan" [Ebrahimi 1994, 459].

Schemata können neben Code-Fragmenten zur Implementierung auch spezielle Programmierepisoden oder Beispiele angelagert sein, sowie Strategien zum Testen und Debuggen[7] assoziieren [vgl. Linn & Dalbey 1989, 61]. Diese Wissensorganisation in Form von untereinander eng vernetzten abstrakten Schemata ermöglicht es Experten,

---

[6] Die Begriffe Schema und Plan werden in der Literatur teils synonym, teils mit unterschiedlicher Betonung der Dynamik verwendet [vgl. Rogalski & Samurçay 1990]. Hier wird der jeweils von den Autoren genutzte Begriff verwendet, aber generell beide als synonym verstanden sein.

[7] Debuggen (Entwanzen) bezeichnet den Prozess des Lokalisierens und Entfernens von Fehlern aus einem Programm [vgl. Balzert 1999, 170ff]. Im Gegensatz zum Testen, dem (systematischen) Prüfen auf Fehlerfreiheit, ist ein Fehler also offensichtlich aufgetreten.

auch bei ungewöhnlichen und komplexen Problemen einzelne Teilziele zusammenzubringen und zu implementieren [vgl. Ormerod 1990, 65f].

Schemata spielen ebenso eine Rolle im Prozess des Programmverstehens, einer Fähigkeit, die für viele Teilaufgaben unerlässlich ist und sich deutlich von der aktiven Generierung unterscheidet [vgl. Robins et al. 2003, 142ff]. Sowohl zum Finden von Fehlern als auch zur Erweiterung bestehender Module muss ein Programm durch innere Simulation seines Ablaufs nachvollzogen werden [vgl. Robins et al. 2003, 150]. Auch wenn unterschiedliche Auffassungen darüber existieren, wie Experten Programme lesen, werden die dabei ablaufenden kognitiven Vorgänge zumeist als Konstruktion mentaler Modelle beschrieben. Mentale Modelle sind eine komplexe Form der inneren Repräsentation eines bestimmten Realitätsausschnitts, die Menschen gerade bei „relativer Komplexität und geringer Transparenz“ [Edelmann 2000, 160] konstruieren, um handeln und neue Informationen einordnen zu können [vgl. Edelmann 2000, 160f; Ben-Ari 1998, 259].

> „Mentale Modelle erlauben die innere Simulation äußerer Vorgänge. Unter den Bedingungen einer bestimmten Anforderungssituation bestimmen sie unser Denken und Handeln. Sie sind subjektive Wissensgefüge und haben einen funktionalen Charakter, d.h. sie ermöglichen die Bewältigung von Aufgaben und Problemen“ [Edelmann 2000, 161].

Eine Erklärung, wie diese mentalen Modelle eines Programms konstruiert werden, bieten Theorien zu 'bottom-up'-Strategien, in denen einzelne Code-Segmente nach und nach als bekannte Muster ähnlich den beschriebenen Plänen erkannt werden und daraus letztlich die Funktion abgeleitet wird [vgl. Grabowski & Pennington 1990, 54]. Brooks [1983 in Grabowski & Pennington 1990] hingegen geht von einem 'top-down'- Prozess des Testens von auf Domain- und Programmierwissen basierenden Hypothesen gegen den Code aus. Allein der Programmname kann hier bereits ein Schema aktivieren und Hypothesen anregen, die dann solange schrittweise verfeinert oder verworfen werden, bis die letzte These gegen 'beacons', typische Indikatoren für eine bestimmte Funktion/ Prozedur, gematcht werden kann [vgl. Grabowski & Pennington 1990, 54].

> „Without a doubt, both top-down and bottom-up processes are involved (…). For example, matching program plan knowledge to code allows the programmer to make inferences about the goals of a program. This in turn leads to further predictions about program contents, a 'top-down' process. In contrast, mental simulation of program effects is a 'bottom-up' process that enables the programmer to reason about the goal-code relations” [Grabowski & Pennington 1990, 54f].

Fix et al. [1993] schließen aus ihrer Untersuchung zum Programmverstehen, dass die mentale Programmrepräsentation eines Experten viele miteinander verbundene Ebenen unterschiedlichen Abstraktionsgrades aufweist, die auf der Erkennung typischer Muster

beruhen und stark im Code verankert sind. In Anlehnung an Ergebnisse der Forschung zum Textverstehen schlägt Pennington [1987] vor, dass sowohl ein Programmmodell als auch ein Modell der Domäne – also über das Problem, das Programm löst, [vgl. Ramalingam & Wiedenbeck 1997, 125] – geformt wird. Dabei kann das Verständnis der Funktion durch das Vorwissen beeinflusst werden, das man über die Problemdomäne besitzt [vgl. Ramalingam & Wiedenbeck 1997, 126]. Das Programmmodell beruht auf elementaren Operationen und dem Kontrollfluss, während das Domainmodell anhand von Datenfluss und Funktionen den Anwendungszweck repräsentiert [vgl. Grabowski & Pennington 1990, 55; Ramalingham & Wiedenbeck 1997, 125ff]. Während des Testens und Debuggens sind neben dem Verstehen zusätzliche Strategien zur systematischen Analyse sowie Wissen über Symptome, Ursachen und deren Korrektur nötig [vgl. Winslow 1996, 20f; Grabowski & Pennington 1990, 55f].

Dieses Expertenwissen als Ziel des Lernprozesses verstanden lässt auf die vielen Herausforderungen schließen, denen sich Programmieranfänger stellen müssen. Im Folgenden wird auf die Charakteristika des Lern- und Problemlöseprozesses eines Anfängers eingegangen.

2.2 PROGRAMMIEREN LERNEN

Das Erlernen einer Programmiersprache stellt für Anfänger oft eine große Hürde dar. Du Boulay [1989] nennt als Ursache dafür fünf Problemfelder, die es gleichzeitig zu bewältigen gilt:

(1) Generelle Orientierung, wozu man Programme erstellt und welche Vorteile man aus ihnen ziehen kann,

(2) „the notional machine“ [Du Boulay 1989, 283], das Entwickeln eines adäquaten Models des Computers bezüglich der Vorgänge bei der Verarbeitung von Programmen[8],

(3) Notation, die Syntax und Semantik einer konkreten Sprache,

(4) Strukturen, d.h. Schemata im Sinne wiederkehrender Code-Muster zur Erreichung bestimmter Pläne und

(5) die Pragmatik des Programmierens, die Fähigkeit Programme ggf. mithilfe verschiedener Tools zu planen und zu entwickeln, zu testen und zu debuggen [vgl. Du Boulay 1989, 283f; vgl. Robins et al. 2003, 148].

[8] Damit ist die 'notional machine' nicht mit dem Computer gleichzusetzen, sondern ist zu sehen als „an idealized, conceptual computer whose properties are implied by the constructs in the programming language employed“ [Du Boulay et al. 1989, 431].

Aufgrund der vielfältigen Herausforderungen stellt Du Boulays 'generelle Orientierung' eine wichtige Motivationsgrundlage dar. Denn gemäß dem pädagogischen Konstruktivismus ist das Erkennen von Praxisrelevanz und Bedeutsamkeit eines Themas – neben der Anschlussfähigkeit neuer Information an die bestehenden kognitiven Strukturen – ein wichtiger Faktor für ein sinnvolles, nachhaltiges Lernen [vgl. Siebert 2003, 24]. Den Nutzen des Programmierens, das automatisierte, effiziente und zuverlässige Lösen bestimmter Probleme, an realistischen Anwendungsfällen zu vermitteln, ist demnach für die Motivation und den Wissenserwerb durch Verankerung im vorhandenen Wissensnetz [vgl. Siebert 2003, 78] eine wichtige Vorraussetzung.

Bereits der Versuch, ein vorgefertigtes Beispielprogramm auszuführen und zu verstehen oder ein eigenes kleines Programm zu erstellen, verlangt von einem Anfänger im Falle von Java den Umgang mit Editoren, Compiler und Interpreter oder sogar einer mächtigen integrierten Entwicklungsumgebung. In der Regel verfügt ein Programmieranfänger dabei über kein adäquates Modell des Computers und kennt die in der 'black box' ablaufenden Mechanismen und Prozesse nicht [vgl. Du Boulay 1989, 285ff; Ben-Ari 1998, 259]. Daher wird er sich aus seinen direkten Erfahrungen und seinen bestehenden kognitiven Strukturen mentale Modelle der Maschine erzeugen [vgl. Du Boulay 1989, 285ff; Ben-Ari 1998, 259].

Bei der Abbildung seiner Funktionsstruktur den Computer und die Programmierumgebung als ein nach strikten Regeln arbeitendes System zu sehen, fällt Anfängern schwer [vgl. Du Boulay 1989, 287]. Mit Interpretationen wie „”it was trying to …”, ”it thought you meant …““ [Du Boulay 1989, 287] werden dem System als einem „giant brain“ [Ben-Ari 1998, 259] vielmehr menschliche Eigenschaften wie das Verstehen der Absicht des Programmierers zugesprochen [vgl. Du Boulay 1989, 287]. Unter anderem wird dadurch die Ausbildung eines Bewusstseins darüber, was ein Programm ist, wie es sich aus seinen Teilen zusammensetzt, wie das System unter Zusammenwirken seiner Komponenten Algorithmen verarbeitet und letztlich auch, wie der eigene Code umgesetzt wird, erschwert [vgl. Du Boulay 1989, 284f].

Um die Konstruktion eines adäquaten mentalen Modells zu erleichtern, sollte ein abstraktes aber effektives Modell des Computers inklusive der 'notional machine' einer bestimmten Programmiersprache explizit gelehrt und diskutiert werden[9] [vgl. Ben-Ari

[9] Dies ist möglich, da Konventionen wie Syntax und Semantik in der Informatik ein normatives Modell bilden, dessen Nichteinhaltung unmittelbar 'bestraft' wird [vgl. Ben-Ari 1998, 259].

1998, 260; vgl. Du Boulay 1989, 285]. Negatives Feedback in Form kryptischer Compiler-Meldungen oder Ausnahmen zur Laufzeit lässt sich dann leichter verstehen und ermöglicht eine sinnvolle Anpassung des eigenen Wissens von Programmierkonzepten und Sprachkonstrukten [vgl. Ben-Ari 1998, 259; Rogalski & Samurçay 1990, 164f].

Das Verstehen der 'notional machine' beeinflusst demnach den Erwerb von Kenntnissen über Syntax und Semantik einer Programmiersprache, auf dem der Schwerpunkt von Einführungskursen liegt. Als Grundlage für das eigenständige Problemlösen werden die entsprechenden Regeln und Konzepte an Beispielen allmählich theoretisch erarbeitet und in praktischen Übungen vertieft [vgl. Robins et al. 2003, 141]. Da heute weit über 200 verschiedene Programmiersprachen existieren, die bezüglich Notation, Sprachumfang und Problemlösungsstil erheblich variieren, stellt jede Sprache ihre eigenen Herausforderungen an die Anfänger [vgl. Balzert 1999, 77f]. Viele dieser Schwierigkeiten könnten durch eine ergonomische Syntax und einen eingeschränkten Sprachumfang gemildert werden [vgl. Hoc & Nguyen-Xuan 1990, 140; Green 1990a, 24]. Jedoch gibt ein spezielles Anwendungsgebiet oder die Verbreitung einer bestimmten Sprache zumeist vor, welche Programmiersprache gelernt wird.

Das Aneignen syntaktischer Regeln, zulässiger Zeichen und Regeln für ihre Kombination, ist in erster Linie ein 'Auswendiglernen'. Da keine vertiefende, bedeutungsvolle Verarbeitung stattfindet, muss dieses Wissen häufig wiederholt angewendet werden, um es im Langzeitgedächtnis zu halten [vgl. Shneiderman 1986, 5]. Ohne die Bedeutung der Zeichen und Elemente, also die Semantik einer Programmiersprache, zu kennen, bleibt dieses Wissen jedoch nutzlos. Es gilt demnach gleichzeitig (großteils) programmiersprachenunabhängige Konzepte wie Variable, Initialisierung, Zuweisung, Bedingung, Iteration, Eingabe, Ausgabe oder Objekte mit den zugehörigen Sprachkonstrukten verstehen und anwenden zu lernen [vgl. Shneiderman 1986, 5].

> "By contrast, semantic knowledge is meaningfully acquired by reference to previous knowledge, by example, or by analogy. There is a logical structure to semantic knowledge that is independent of the specific syntax used to record it" [Shneiderman 1986, 5].

Zur Lösung einer Problemstellung setzt ein Anfänger sein deklaratives Faktenwissen über Sprachnotation und Konzepte ein, um sich Zeile für Zeile dem Ziel zu nähern. Denn er verfügt nicht oder nur in geringem Maße über programmierspezifische Pläne und Produktionsregeln zur systematischen Dekomposition eines Problems in implementierbare Teilziele. So verbringen Novizen generell wenig Zeit mit dem Planen bzw. Entwerfen eines Programms, sondern identifizieren Ziele und Pläne 'on the fly' [Lane &

VanLehn 2004, 449] während der Implementierung, statt in einer separaten Planungsphase [vgl. Robins et al. 2003, 151]. Ihr Planen ist ein opportunistischer Prozess, in dem sie vom Ziel des Programms her rückwärts denkend nötige Zwischenziele 'depth-first' nacheinander ermitteln, zeilenweise konstruieren und ihre Schritte regelmäßig am Problem evaluieren und verwerfen [vgl. Anderson 2000, 294ff; Davies 1993]. Nach erfolgreicher wiederholter Anwendung bilden die durch dieses 'goal back-chaining' [Rist 1986, 30] entstandenen Code-Fragmente für ein Teilziel dann besagte Pläne [vgl. Rist 1986, 31].

> „The goal chain defines the overall structure of the program. Novices focus on construction issues and use this goal information to implement program operators. They work from the code to the goals. As templates form, these goals index code segments and code is retrieved from the plan library by the goal that it realizes" [Rist 1986, 31f].

Neben der fast ausschließlichen Orientierung am Ziel nutzen Anfänger zusätzlich verstärkt allgemeine Problemlösestrategien wie das Übertragen von Lösungen früherer Fälle oder Analogieschlüsse zu anderen Domänen [vgl. Winslow 1996, 18; Bonar & Soloway 1989, 325]. „Problem solution by analogy is common at all levels; choosing the proper analogy may be difficult" [Winslow 1996, 18].

Auch beim Lesen von Beispielprogrammen oder dem Suchen von Fehlern im eigenen Programm unterscheiden sich Anfänger stark von Experten. Ein effektives Erfassen eines Programms auf mehreren zunehmend abstrakteren, miteinander vernetzten Ebenen – vom konkreten Code über Code-Muster hin zum Programmziel – weisen die mentalen Programmmodelle der Anfänger (1. Semester Pascal) bei Fix et al. [1993] kaum auf. Als Gründe vermuten die Autoren fehlendes Wissen bspw. über Pläne, eine andere Lesereihenfolge als dem Kontrollfluss folgend (z. B. von links nach rechts abwärts wie beim normalen Lesen [vgl. Rogalski & Samurçay 1990, 165]) und Mangel an Fähigkeit zur symbolischen Programmausführung [vgl. Fix et al. 1993, 78f]. Auch hier orientieren sich Anfänger demnach vor allem an oberflächlichen Merkmalen wie der Syntax, statt ein Programm als eine Folge von Funktionen mit Teilzielen zu verstehen.

Eine erste Programmiersprache zu lernen bedeutet folglich nicht nur, Wissen über die Notation der Sprache und ihre Konzepte aufzubauen, sondern auch ein effektives Model der 'notional machine', Strukturen in Form von Schemata, und Strategien zum algorithmischen Problemlösen, Planen und Debuggen zu erwerben.

> „(M)uch of the 'shock' of the few first encounters between the learner and the system are compounded by the student's attempt to deal with all these different kinds of difficulty at one" [Du Boulay 1989, 284].

Die teilweise bereits angedeuteten Problembereiche und daraus resultierende Fehler werden nun ausführlicher betrachtet.

2.3 ANFÄNGERPROBLEME

Um den Lernprozess von Anfängern besser verstehen und unterstützen zu können, untersuchen viele Studien Anfängerfehler und -probleme und ihre möglichen Ursachen [Soloway & Spohrer 1989, 230f]. Da die verschiedenen Untersuchungen hinsichtlich ihrer zentralen Fragestellung, Methodik, ihrer Terminologie und Klassifikation stark variieren, werden hier Fehler umfassend für einen nicht ausführbaren oder einen nicht gemäß den Spezifikationen der Aufgabe funktionierenden Code verwendet[10]. Studien zum Programmverstehen berichten zudem von fehlerhaften Annahmen über Code-Segmente oder eine falsche Vorhersage ihrer Ausführung, wobei hier von Missverständnissen bzw. Misskonzeptionen gesprochen werden soll. Dabei werden nur solche Fehler betrachtet, die in Folge eines Denkfehlers entstanden sind [vgl. Grams 1990, 17ff; Reason 1994, 129f]. Derartige Fehler können aufgrund fehlenden oder falsch angewandten Wissens über Syntax und Semantik einer Programmiersprache entstehen, die in Java häufig eine Compiler-Meldung oder möglicherweise einen Laufzeitfehler hervorrufen. Fehler können aber auch Folge einer unzureichenden Problemanalyse und mangelhafter Spezifikationen sein. Neben semantischen Missverständnissen kann dies zu Programmen führen, die zwar ausführbar sind, ihre Aufgabe aber nicht vollständig oder nicht korrekt erfüllen [vgl. Ko & Myers 2003, 8]. Diese beiden Problembereiche – das Beherrschen einer bestimmten Programmiersprache und das strategische Problemlösen durch Algorithmen – sollen nun nacheinander betrachtet werden.

2.3.1 Programmiersprachliche Notationen und Konzepte

Obwohl Programmiersprachen je nach Notation und Sprachumfang variieren und Anfänger diese Regeln oft missachten, berichten nur wenige Studien in größerem Umfang von Syntaxfehlern. Zu Erwähnen ist dabei, dass die Anzahl an Untersuchungen zum Programmverstehen weit höher ist [vgl. Robins et al. 2003, 144] und einige Studien zur aktiven Generierung als Datenmaterial nur bereits syntaktisch korrekte Programme heranziehen. So nennt Du Boulay [1989] in seinen Betrachtungen zu den Schwierigkeiten

[10] Im Englischen lassen sich gemäß IEEE Standard 610.12-1990 für Softwarequalität 'error', 'fault' und 'failure' unterscheiden [nach Ko & Myers 2003, 7]. Im Deutschen ist die Trennung unschärfer, da 'Fehler' oder 'Fehlverhalten' oft sowohl für die Entstehung als auch das Resultat genutzt werden.

des Programmierlernens als rein syntaktisches Problem allein das Vergessen oder inkorrekte Einfügen des Semikolons in Folge von Übergeneralisierung „from one part of the language to another“ [Du Boulay 1989, 298]. Ebrahimi [1994] weist in seiner vergleichenden Studie zu Sprachkonstrukten in LISP, C, Pascal und Fortran unter anderem auf bestimmte ungünstige Notationen in C – wie bitweises vs. logisches 'AND' bzw. 'OR' und '==' als Gleichheitsoperator – hin. Eine Verwechslung kann in diesen Fällen zu einem Programm führen, das zwar syntaktisch korrekt ist, jedoch nicht seine Spezifikation erfüllt.

Beide Notationen kommen auch in der Programmiersprache Java vor und werden von Hristova et al. [2003] unter den charakteristischen Java-Anfängerfehlern geführt. Die Autoren ermittelten bei einer Umfrage unter Studenten und Dozenten die 20 häufigsten Java-Fehler und klassifizierten sie aus der Sicht eines Anfängers (vs. Compiler) als Syntaxfehler, semantische oder logische Fehler. Als andere prominente Syntaxfehler nennen die Autoren '==' vs. 'equals' bei Stringvergleichen, Klammersetzung allgemein, Setzen des Semikolons, falsche Trennzeichen innerhalb der 'for'-Schleifendeklaration, Methodenaufruf ohne Klammern, Vergeben von Schlüsselwörtern als Variablen- oder Methodennamen, eine falsche Reihenfolge in Vergleichen wie '>=' und das Aufrufen einer Methode mit falschen bzw. fehlenden Parametern.

Zumindest bei letztgenanntem Fehler ist fraglich, ob es sich hier 'nur' um einen Syntaxfehler handelt oder nicht ein generelles Verständnisproblem von Methoden und Parameterübergabe zugrunde liegt. Orientiert man sich am Lernenden und nicht am Compiler, scheint eine klare Trennung zwischen beiden Kategorien – abgesehen von Klammer- und Semikolonsetzung – kaum möglich. Vielmehr zeigen frühe Studien, die vor allem den Einfluss bestimmter Sprachkonstrukte wie 'if-then-else' bzw. verschachtelte 'if'-Anweisungen und 'GOTO's zwischen verschiedenen Sprachen auf ihre Fehleranfälligkeit hin untersuchten [siehe Sheil 1981; vgl. Soloway & Spohrer 1989; siehe Robins et al. 2003, 147], dass die Syntax nicht von der Semantik zu trennen ist. Vielmehr sind beide als komplementär zu betrachten und so manifestieren sich in Syntaxfehlern oft Verständnisschwierigkeiten [vgl. Hoc & Nguyen-Xuan 1990, 140]:

> „Syntactic errors have been shown to be of limited importance even to beginners. … In a number of studies on learning to program, which stress the acquisition of a new communication means, semantic rather than syntactic difficulties are shown. This is especially true of Soloway’s works which show that beginners introduce distortions into programming language syntax when their programming knowledge is lacking. These distortions are indicators of transfers from other

> knowledge domains that are not compatible with the programming language structure" [Hoc & Nguyen-Xuan 1990, 140][11].

Eine wesentliche Quelle negativen Transfers bildet dabei die natürliche Sprache bzw. das 'Preprogramming knowledge' [Bonar & Soloway 1989]. Oberflächliche Ähnlichkeiten zur natürlichen Sprache können demnach zu Missverständnissen in Bezug auf bestimmte Sprachkonstrukte führen, wenn bspw. in einer 'while'-Schleife in Analogie zur Semantik des natürlichsprachlichen 'while' erwartet wird, dass die Endbedingung kontinuierlich – und nicht nur bei jedem neuen Schleifendurchlauf – vom System überwacht wird [vgl. Bonar & Soloway 1989, 332]. „'Repeat' misleads beginners who expect that there must be something already in existence to be repeated, and so on." [Du Boulay 1989, 288]. Das Nachvollziehen des Kontrollflusses innerhalb eines Programms über 'then' ('if-then') oder den Boole'schen Operator 'AND' wird nach Du Boulay [1989, 288ff] durch solche Assoziationen erschwert, da beide in der natürlichen Sprache oft als '*danach*' verwendet werden. Probleme durch Verwechslungen mit natürlicher Sprache im Falle der Boole'schen Operatoren 'OR'-'AND' beschreiben ebenso Soloway & Spohrer [1986a] als 'natural language problem'.

Das 'Raten' von Sprachelementen in Anlehnung an die natürliche Sprache oder den 'lockeren' Umgang mit der Programmiersprache begründen Soloway & Spohrer [1986a, 631] als 'human interpreter problem' mit der Erwartung, dass die Intention – auch unter Nichtbeachtung der strikten Regeln von Syntax und Semantik – vom System verstanden wird. Für Martin & Perkins [1986] drückt sich darin der Wille der Anfänger aus, angesichts nicht ausreichender Kenntnisse das Problem dennoch zu lösen.

Nicht nur die natürliche Sprache bedingt negativen Transfer. Auch Vorwissen verschiedener Bereiche fließt in die Konstruktion der mentalen Modelle darüber, welche unsichtbaren Vorgänge im Inneren des Computers infolge eines bestimmten Code-Fragmentes ablaufen, mit ein [vgl. Bayman & Mayer 1983, 677ff; vgl. Du Boulay 1989, 284]. „Although prior knowledge is of course an essential starting point, there are times when analogies applied to the new task of programming can also be misleading" [Robins et al. 2003, 152; vgl. Hoc & Nguyen-Xuan 1990, 140]. Und so berichten viele Studien – unabhängig von einer speziellen Programmiersprache – von fehlerhaften Vorstellungen von Konzepten wie Variable, Initialisierung, Zuweisung, Bedingung und vor

[11] Dieser Effekt negativen Transfers konnte auch bei Experten gezeigt werden, die beim Lernen einer weiteren Programmiersprache ihnen bekannte Konzepte nicht in der neuen Sprache umsetzen konnten [vgl. Hoc & Nguyen-Xuan 1990, 140].

allem der Iteration und den Unterschieden zwischen einzelnen Schleifentypen [siehe Robins et al. 2003, 151f; siehe Soloway & Spohrer 1989].

Du Boulay [1989, 283f] sieht die wesentliche Ursache von Missverständnissen in falschen Theorien bzw. Modellen von der 'notional machine' und der damit einhergehenden Anwendung unpassender Analogien auf Programmierkonzepte wie Variablen, Zuweisung und Arrays. Variablen werden häufig als eine Art Box oder beschreibbare Schiefertafel begriffen, was der Variablen als Speicherort insofern nicht gerecht wird, als dass nicht mehrere Werte gleichzeitig 'aufbewahrt' werden [Du Boulay 1989, 291]. Der Vergleich mit einer Box könnte erklären, warum bei dem Vertauschen der Werte zweier Variablen das Zwischenspeichern des ersten Variablenwertes zumeist vergessen oder die Reichweite von Variablenwerten nicht beachtet wird [Du Boulay 1989, 291f]. Die Box-Metapher kann ebenso für das typische Vergessen von Initialisierungen aufkommen. Denn solange man nichts in eine Box getan hat, „it's empty, which is sort of like zero" [Du Boulay 1989, 292]. Die Schwierigkeiten mit den Konzepten Variable und Initialisierung zeigt auch Samurçay [1989] auf. Fehler in Bezug auf Initialisierungen traten in Ebrahimi's [1994] vergleichender Studie zwischen C, Lisp, Fortran und Pascal über alle Sprachen hinweg häufig auf.

Die Referenzen auf 'versteckte' Speicherplätze ohne erfahrbare physikalische Entsprechung und die damit verbundene Unterscheidung zwischen Variablen als Speicherorten und/ oder konkreten Werten bedingen nach Du Boulay [1989, 293] weitere Missverständnisse im Umgang mit Zuweisungen und mit komplexen Datentypen wie Arrays. Weitere Bespiele für Verständnisprobleme, die sich dem Autor nach auf mangelnde Kenntnis und falsche Vermutungen über die unsichtbaren Prozesse zurückführen lassen, betreffen Objektgleichheit vs. -identität, Datentyp vs. Wert einer Variablen, Schleifen und Kontrollfluss [Du Boulay 1989, 291-295]. Von Anfängern wird häufig übersehen, dass während ein Programm abläuft, gemäß den Anweisungen schrittweise Zustände von Variablen geändert und jede Anweisung in dem Zustand ausgeführt wird, der durch die vorhergehenden bestimmt wurde [Du Boulay 1989, 294].

Die „hidden, internal changes" [Du Boulay 1989, 295] begünstigen zum einen fehlerhafte Schleifen (update der Laufvariable, Endbedingung), wie sie auch Ebrahimi [1994] als häufige Fehler (LISP, Fortran, Pascal) in seiner Studie ermittelte. Sie können aber auch Missverständnisse während der Ausführung bedingen, wenn bspw. bei einer 'READ'-Anweisung die Ausführung des Programms stoppt bis eine Eingabe vom Be-

nutzer erfolgt ist, dieser jedoch wartet, dass das Programm weiterläuft [vgl. Du Boulay 1989, 296]. Du Boulays Schlussfolgerung ist daher, dass Anfänger ein Modell der 'notional machine' an die Hand bekommen müssen, anhand dessen versteckte Prozesse aufgezeigt werden können und auf die Gefahr ungeeigneter Analogien hingewiesen werden kann.

Bei dem Lernen einer objektorientierten Programmiersprache wie Java trifft ein Anfänger zusätzlich auf das Konzept 'Objekt' mit seinen Prinzipien der Datenkapselung, Vererbung und Polymorphismus. Trotz des Anspruches, dass das Modellieren und Arbeiten mit Objekten dem menschlichen Denken und seiner Wissensorganisation angemessener ist [vgl. Green 1990a, 25; siehe Détienne 1997], scheinen die wenigen Studien zu objektorientierten Prinzipien zumindest in Bezug auf Anfänger keine Erleichterung feststellen zu können [siehe Détienne 1997]. Basierend auf eigenen Beobachtungen in der Lehre weisen Griffiths et al. [1997, 131ff] auf die Gefahr des Missverstehens von Objekten hin. Konfusion zwischen Objekten, Instanzvariablen und Klassen, und damit verbunden auch Objektgleichheit vs. -identität, sind neben einer statischen Interpretation von Objekten als reine Datenbankeinträge, also ohne zustandsabhängiges Verhalten und Kommunikation, typische Fehlerquellen. Griffiths et al. [1997, 131ff] betonen daher die Wichtigkeit einer gezielten Auswahl von Beispielen und Aufgaben in Kursen zur objektorientierten Programmierung.

Détienne [1997] berichtet ebenfalls von Problemen mit Instanzvariablen, nämlich der Annahme, dass diese automatisch erzeugt werden und von Verständnisschwierigkeiten bezüglich Vererbung. Auch Bancroft et al. [2004, 319f] weisen neben dem Vergessen von 'default' oder 'break'-Anweisungen bei 'switch-case', den fehlenden Aufruf von 'super( ) ' im Konstruktor einer abgeleiteten Unterklasse und Verwechslungen zwischen Instanzen und lokalen Variablen als typische logische Anfängerfehler in Java aus. Als häufige Misskonzeptionen fand Fleury [2000] Initialisierung und Speicherplatzallokation eines Objektes mithilfe des Konstruktors, Parameterübergabe bei Methoden und die Anwendung des '.'-Operators heraus.

Ähnlich haben Hristova et al. [2003] in ihrer Umfrage als typischen semantischen Fehler das direkte Aufrufen einer Klassenmethode für ein Objekt bzw. eine Variable statt Benutzung des '.'-Operators ermittelt. Unter logischen Fehlern führen sie das Casten mit Datenverlust bspw. zwischen 'float' und 'int', Verwechslungen zwischen Parameterdeklaration und -übergabe, fehlendes Implementieren abstrakter Methoden einer

Schnittstelle/ Oberklasse, das Nichtverwenden des Rückgabewertes einer Methode, Inkompatibilität zwischen dem Datentyp zum Aufruf und der Rückgabe einer Methode und eine fehlende 'return'-Rückgabeanweisung in 'non-void'-Methoden.

Probleme mit dem Verständnis von 'return', gerade in Bezug auf den Kontrollfluss innerhalb von Schleifen, fanden auch Adams et al. [2004] in ihrer internationalen Studie, in der Studenten die Ausführung von Code-Fragmenten vorhersagen und sie zusätzlich komplettieren sollten. Jedoch war dies ihrer Auffassung nach das einzige auf Sprachkonstrukte zurückführbare Missverständnis. Dies steht im Einklang mit den vielen Studien, welche Anfängerprobleme und -fehler vor allem auf mangelnde programmierspezifische Problemlösetechniken zurückführen [vgl. Winslow 1996, 17f].

2.3.2 Algorithmisches Problemlösen

Während die Nichteinhaltung syntaktischer Regeln, missverstandene Sprachkonstrukte und ungeeignete mentale Modelle von den Konzepten einer Programmiersprache Ursachen von Fehlern sein können, stellt sich für Anfänger in der Generierung von Programmen vor allem die Schwierigkeit die einzelnen Teile zielgerichtet zusammenzufügen.

> „An example in point is the large number of studies concluding that novice programmers know the syntax and semantics of individual statements but they do not know how to combine these features into valid programs. Even when they know how to solve a problem by hand, they have trouble translating the hand solution into an equivalent computer program" [Winslow 1996, 17].

Da die Erstellung eines Programms im ersten Schritt die Dekomposition der Problemstellung in algorithmisch berechenbare Teilziele voraussetzt, Anfängern die Kenntnisse über die jene Teilziele realisierenden Pläne jedoch fehlen, entstehen viele Fehler „as the novice tries to put the 'pieces' of a program together in the Plan Composition"[Ebrahimi 1994, 460; Soloway & Spohrer 1986a, b]. Auf Basis der in zwei Studien analysierten Programmen von Pascalanfängern[12] [Soloway & Spohrer 1986a, b] benennen Soloway & Spohrer [1986b] unter anderem solche 'plan composition problems':

> „(a) *Summarisation problem*. Only the primary function of a plan is considered, implications and secondary aspects may be ignored.
> (b) *Optimisation problem*. Optimisation may be attempted inappropriately.
> (c) *Previous-experience problem*. Prior experience may be applied inappropriately.
> (d) *Specialisation problem*. Abstract plans may not be adapted to specific situations.
> (e) *Natural-language problem*. Inappropriate analogies may be drawn from natural language.

[12] Die Autoren analysierten zu drei verschiedenen Aufgabenstellungen von 61 Studenten jeweils die erste syntaktisch korrekte Version eines Programms. Fehler wurden gemäß ihrer Theorie von Programmzielen und -plänen anhand der Unterschiede zwischen den implementierten Plänen der Studenten und korrekten Plänen für eine Aufgabe ermittelt. Damit sind neben falschen Methodenaufrufen auch vergessene Anweisungen oder Anweisungen an einer unpassenden Stelle ein Fehler.

(f) *Interpretation problem*. ‘‘Implicit specifications’’ can be left out, or ‘‘filled in’’ only when appropriate plans can be easily retrieved.
(g) *Boundary problem*. When adapting a plan to specific situations boundary points may be set inappropriately.
(h) *Unexpected cases problem*. Uncommon, unlikely, and boundary cases may not be considered.
(i) *Cognitive load problem*. Minor but significant parts of plans may be omitted, or plan interactions overlooked” [gekürzte Zusammenstellung nach Robins et al. 2003, 153].

Die meisten dieser Probleme stehen in Zusammenhang mit der Anwendung allgemeiner Problemlösetechniken, die Anfänger zu Überwindung von Problemsituationen zur schrittweisen Annäherung ans Ziel einsetzen. Hierzu zählt das Heranziehen von Beispielen oder früheren Lösungen, wobei sich Anfänger jedoch stark an oberflächlichen Merkmalen der Problemdomäne oder syntaktischen Ähnlichkeiten orientieren [vgl. Grabowski & Pennington 1990, 48f; vgl. Winslow 1996, 18]. Während Experten Problemstellungen gemäß den erforderlichen algorithmischen Problemlöseschritten vielfältig repräsentieren und umformulieren können [vgl. Grabowski & Pennington 1990, 48f; vgl. Winslow 1996, 18], gelingt es Anfängern kaum, für ein Ziel geeignete Beispiele auszuwählen bzw. nötige Pläne selbst zu konstruieren.

Nach Martin & Perkins [1986] ist die Ursache dafür weniger in fehlendem Wissen oder Misskonzeptionen als vielmehr in vorhandenem aber nicht in der geeigneten Situation zugänglichem 'inert knowledge' zu sehen. In Folge mangelnder Abstraktion scheinen diese Kenntnisse nicht über den Kontext des ursprünglichen Lernens hinaus auf neue Kontexte übertragbar zu sein. In einer Untersuchung von 20 Highschool-Schülern in ihrem ersten Jahr BASIC-Unterricht führen die Autoren 47 % aller bei der Aufgabenbearbeitung aufgetretenen Schwierigkeiten auf nicht einsetzbares 'inert knowledge' zurück. Durch strategische Nachfragen[13] und Hinweise des Experimentleiters gelang es in diesen Fällen, das korrekte Wissen abzurufen und die Aufgabe zu erfüllen.

Dies erklären Martin & Perkins [1986] damit, dass das Wissensnetz eines Anfängers weniger bzw. weniger starke Verlinkungen zwischen verschiedenen Zielen, Plänen und Anweisungen aufweist und somit die Zugänge für den Abruf beschränkt sind. Von allein wandten die Anfänger generelle Strategien wie Planen, Hinterfragen und Umformulieren des Problems kaum an. „(T)he strategic shortfall exacerbates the fragile knowledge problem“ [Martin & Perkins 1986, 226]. Schwierigkeiten, Pläne und Ziele zuzuordnen, spiegeln sich ebenso in dem von Martin & Perkins [1986] und von Soloway and Spohrer [1986b] beobachteten 'Recency effect', der Annahme, dass kürzlich Gelerntes

[13] Als strategische Nachfragen dienten bspw. „What is your plan?“ oder „Do you know a command for repeating?“[vgl. Martin & Perkins 1986, 223]

wieder anwendbar sein müsse. Unter 'misplaced knowledge' fassen Martin & Perkins [1986] korrektes Faktenwissen, das sich aber in einer konkreten Situation nicht zur Erreichung eines Zieles eignet, und benennen als Gründe bspw. eine solche Übergeneralisierung.

Die natürliche Sprache ist nach Bonar & Soloway [1989, 331] nicht nur Quelle für negativen Transfer im Hinblick auf einzelne Sprachkonzepte, sondern auch bei der Identifizierung von Zielen und der Konstruktion von Plänen. Diese Pläne zur Erreichung bestimmter Ziele in der natürlichen Sprache weisen starke funktionelle Verbindungen zu den entsprechenden Plänen einer Programmiersprache auf und sind damit für die Problemlösung hilfreich [Bonar & Soloway 1989, 331]. Novizen überschätzen häufig jedoch aufgrund oberflächlicher Ähnlichkeiten zur englischen Sprache (Methodennamen, Schlüsselwörter) ihre Übertragbarkeit auf das Programmieren: "novices inappropriately using their knowledge of step-by-step procedural specifications in natural language" [Bonar & Soloway 1989, 325]. So lässt die natürliche Sprache viele Informationen implizit, die ein (menschlicher) Adressat aus seinem Hintergrundwissen und dem Kontext erschließen kann. Als ein Beispiel zur Illustration diene die mündliche Ausführung einer Aufgabe, in der die Gehaltchecks aller Mitarbeiter geprüft werden sollen: „"1. Identify worker, check name on list, check wages." Here … getting the input value (the worker) has been done implicitly" [Bonar & Soloway 1989, 333].[14] Dass ein Algorithmus hingegen mit allen nötigen Einzelschritten zur Datenverarbeitung detailliert und formalisiert implementiert werden muss, ist für Anfänger nicht selbstverständlich [vgl. Du Boulay 1989, 288].

Das Auslassen entscheidender Pläne in einem Programm kann im Falle von nicht implementierten oder unzureichenden 'If'-Bedingungen mithilfe des 'Scheinwerferprinzips' erklärt werden. Grams [1990, 36f] beschreibt unter diesem Begriff die generelle menschliche Neigung – auch bei Experten –, gemäß der selektiven Wahrnehmung unerwartete Grenzfälle außer Acht zu lassen. Pope et al. [1989] und Ebrahimi [1994] haben so basierend auf einem aufgabenspezifischen Kategorisierungsschema zu Plandifferenzen[15] in den Anfängerprogrammen das 'missing Guard IF' als den häufigsten Plankom-

[14] In ihrer Studie führen Bonar & Soloway [1989] Interviews mit Programmieranfängern, die eine Aufgabe zuerst mündlich ausführten.

[15] Fehler sind hier das Fehlen eines für die Aufgabenstellung notwendigen Plans 'missing', einem Plan an falscher Stelle 'misplaced', seine fehlerhafte Implementierung 'malformed' sind oder unnötigem Code 'spurious' [Pope et al. 1989].

positionsfehler gefunden[16] [17]. „Students neglected to take into account, when using IF statements, the need to check for special situations, such as division by zero" [Ebrahimi 1994, 473]. Pope et al. [1989] beobachteten zudem, dass zahlreiche Programme 'merged goals'/ 'merged plans' enthielten und damit meist eine höhere Fehlerzahl verbunden war.

> „When we say „goals were merged," we mean that a single plan to achieve these two goals in an integrated manner was used. The merged plan may have been produced by taking into two plans that would achieve the goals independently and combining them in some nontrivial manner" [Pope et al. 1989, 384].

Ein Beispiel wäre das gleichzeitige Einlesen und Verarbeiten von mehreren Daten in einer Methode, statt dem wiederholten Aufruf der Methode oder Prüfen auf mehrere Bedingungen in einer 'If'-Anweisung. Diese Strategie ist beim Problemlösen prinzipiell sehr effizient, „(u)nfortunately, employing the strategy of goal/plan merging in programming requires considerable skill" [Pope et al. 1989, 397]. Zusammengeführte Ziele können zum Auslassen wichtiger Unterziele und komplexen Code-Strukturen führen, die das Nachvollziehen von Abhängigkeiten zwischen Fragmenten und die Erweiterung um Pläne, aber auch die Fehlerfindung erschweren [vgl. Pope et al. 1989, 391ff]. Die Schwierigkeiten der Anfänger, Probleme in vom Computer ausführbare Einzelschritte zu zerlegen und die Teilziele in Code zu übersetzen, bestätigt auch die Selbsteinschätzung der 559 Studenten in der Umfrage von Ala-Mutka et al. [2005].

> „The respondents perceived as the most difficult issues in programming understanding how to design a program to solve a certain task …, dividing functionality into procedures … and finding bugs from their own programs …. These are all issues where the student needs to understand larger entities of the program instead of just some details about it" [Ala-Mutka et al. 2005, 16].

Referenzen, abstrakte Datentypen, Fehlerbehandlung und die Klassenbibliotheken stuften die Teilnehmer als die schwierigsten Programmierkonzepte ein, wobei vielmehr die aktive Anwendung als deren Verstehen Probleme bereitet [vgl. Ala-Mutka et al. 2005, 16f]. Objekte als komplexe, abstrakte Datentypen ausgehend von einer Problembeschreibung zu modellieren und dementsprechend Klassen, Funktionen und Interaktionen eines Programms zu planen, fällt Anfängern der objektorientierten Programmierung schwer [vgl. Berge et al. 2004; vgl. Lavy & Or-Bach 2004]. Lavy & Or-Bach [2004] berichten, dass viele Anfänger in einer Aufgabe zur Erstellung eines Klassendiagramms nicht den nötigen Grad an Abstraktion erreichen, um bspw. Vererbung von Attributen

---

[16] Dieser Fehler trat bei Ebrahimi [1994] in den Programmen aller vier verglichenen Programmiersprachen auf.

[17] Pope et al. [1989] weisen aber darauf hin, dass sie jeweils die erste syntaktisch korrekte Version eines Programms zur Auswertung herangezogen haben und somit das Prüfen nur im ersten Arbeitsschritt von den Studenten weggelassen sein könnte.

und Methoden effizient einzusetzen. „(I)t seems that the major cited advantages of object orientation are exactly the same issues that make object orientation so difficult for students“ [Lavy & Or-Bach 2004, 84]. Zusätzlich führt die Verteilung von Kontrollfluss und Funktion durch Objektorientierung gerade in langen Programmen zu fehlerhaften Modellen des Programmablaufs [vgl. Ramalingam & Wiedenbeck 1997, 132-135], was das Finden von Fehlern erschwert. Zu umfangreiche Methoden und das Verwenden von globalen Variablen statt Parametern, wie sie Bancroft et al. [2004, 319f] als schlechte Programmierpraktiken von Anfängern berichten, dürften das Nachvollziehen von Kontroll- und Datenfluss zusätzlich beeinträchtigen.

Putnam et. al [1989, 312f] sprechen allgemein von mangelnden Fähigkeiten zur systematischen Evaluierung von Programmen und zum Rückverfolgen von Fehlern „(b)y taking the computer's point of view" [Hancock et al. 1989, 268]. Stattdessen wird zumeist nach 'Versuch-und-Irrtum' verfahren [auch Perkins et al. 1988; Martin & Perkins 1986]. „They would trace and predict output, ignoring or misinterpreting statements that did not fit with what they thought the program would do" [Baxter et al. 1989, 312]. Zudem fehlen Anfängern für das Testen und Debuggen erforderliche Strategien wie die sukzessive Überprüfung jeweils einer kritischen Stelle, die Auswahl geeigneter Testwerte oder das Einfügen unzähliger Ausgabeanweisungen [vgl. Winslow 1996, 20f].

„All bugs are not created equal. Some occur over and over again in many novice programs, while others are more rare“[Soloway & Spohrer 1986a, 250]. Fehler entstehen oft im Zusammenspiel von Problemen mit Sprachkonstrukten, inadäquaten mentalen Modellen der 'notional machine' und mangelnden algorithmischen Problemlösestrategien, so dass Fehlerursachen dabei nicht immer eindeutig zuzuordnen sind. Nach Roschelle et al. [1993] nehmen Denkfehler und Missverständnisse aus konstruktivistischer Sicht jedoch eine wichtige Position im Prozess des Lernens ein. Denn erst durch die Integration neuer Inhalte in bestehendes Wissen – bspw. mithilfe nur teilweise geeigneter Analogien – wird ein nachhaltiges Lernen möglich [vgl. Roschelle et al. 1993]. „Good design comes from experience, but experience comes from bad design" [Bergin 2000, zitiert nach Michiels & Börstler 2000]. Feedback und Reflektion sind dafür zentrale Vorraussetzungen. Verschiedene Programmierumgebungen bieten in dem aktiven Problemlöseprozess unterschiedliche Arten von Hilfestellungen und Feedback, welche wiederum hinsichtlich ihres Ausmaßes variieren.

3 PROGRAMMIERUMGEBUNGEN

Kompetenz im Bereich der Programmierung zu erwerben, setzt die praktische Ausübung der theoretisch erworbenen Kenntnisse voraus.

> „Programming demands a great deal of implicit knowledge which is difficult for lecturers to make explicit and which can not easily be transmitted to students. In order to gain knowledge and become competent in the domain, students need to go beyond explicit information to construct experiential implicit knowledge (…). Programming cannot be learnt without a lot of practice" [Bancroft et al. 2005, 1].

Als Mindestausstattung zur eigenen Entwicklung von Programmen wird dazu ein Texteditor zur Eingabe des Codes benötigt, der im Falle von Java mittels Compiler-Aufruf in der Eingabeaufforderung in Maschinen-Code übersetzt und vom Interpreter ausgeführt wird. Spezielle Editoren und integrierte Entwicklungsumgebungen (IDEs), die Editor, Compiler und Interpreter mit Werkzeugen zur Fehlerverfolgung und Dokumentverwaltung kombinieren, ermöglichen eine effizientere Aufgabenbearbeitung über eine zentrale Benutzeroberfläche [vgl. Balzert 1999, 99]. Mit dem Ziel, nützliches Feedback zum Aufbau adäquater mentaler Modelle geben zu können [vgl. Bancroft et al. 2005], leisten automatische Debugger hingegen konkrete Hilfestellung, indem sie Fehler aufdecken und benennen. Intelligente Tutorielle Systeme (ITS) verfolgen den Anspruch, den Lernenden durch Modellierung seiner Fähigkeiten aktiv zu führen oder zumindest durch intelligente Fehlerdiagnose zu unterstützen. Intelligente Entwicklungsumgebungen letztlich versuchen, die Vorteile aus den genannten Plattformen zu vereinen: verschiedene Werkzeuge zum selbständigen Problemlösen während der explorativen Lernphase sowie gezielte tutorielle Unterstützung bei Problemen[18].

Als Einführung in Analyse- und Unterstützungsmöglichkeiten bei der Programmentwicklung werden nun verschiedene Ansätze und Plattformen exemplarisch vorgestellt, um ihre Potentiale im Hinblick auf die Bedürfnisse eines Programmieranfängers aufzuzeigen.

3.1 PROGRAMMIERWERKZEUGE

RealJ (vorher FreeJava) ist eine für die ersten Programmiererfahrungen oft eingesetzte frei erhältliche[19] IDE für Java [Thefreecountry]. Sie umfasst einen Texteditor mit

[18] Da die Entwicklung dieser intelligenten Systeme ihren Höhepunkt in den 1980er und 90er Jahren erfahren hat, enthält der Überblick keine Systeme für Java oder eine andere OO-Sprache, für die bisher nur Prototypen existieren.

[19] Bis zur Version 3.5 besteht die Möglichkeit eines kostenlosen Downloads

'Syntax-Highlighting'[20], einen integrierten Compiler und Interpreter, so dass umständliches Wechseln zwischen den Komponenten zugunsten einer stärkeren Konzentration auf das eigentliche Programmieren vermieden werden kann. Es besteht zudem die Möglichkeit, eine JDK-Hilfe so zu installieren, dass bei Markieren eines Schlüsselwortes oder Klassenamens unter Drücken der Funktionstaste F1 die entsprechende JDK-Information aufgerufen wird.

Eclipse [Eclipse] umfasst als ein mächtiges Entwicklungstool für professionelles Programmieren umfangreiche Funktionen, die sich aufgrund seiner offenen Plugin-Architektur um eigene Komponenten erweitern lassen [vgl. Weyershäuser 2003, 12f]. Aufgrund der vielen Eigenschaften, die für Aufgaben von Anfängern nicht benötigt werden, sehen bspw. Balzert & Balzert [2004, 24f] Eclipse als zu unübersichtlich und eher als eine Überforderung an. Interessant ist jedoch der Einsatz eines inkrementellen Compilers, durch den der Code während der Eingabe in den Editor im Hintergrund kompiliert und vom Compiler angegebene Syntaxfehler im Quelltext markiert werden [vgl. Weyershäuser 2003]. Die 'Quickfix'-Funktionalität unterbreitet zu solchen Fehlern, die über die reine Klammer bzw. Semikolonsetzung hinausgehen – falls möglich – Korrekturvorschläge.

Andere Werkzeuge, die Vorgänge während der Ausführung eines Programms und möglicherweise Fehler leichter nachvollziehbar machen können, bedienen sich der Techniken der Software-Visualisierung (SV). Beispiele für SV-Techniken sind statische Darstellungen von Kontroll- und Datenfluss in Flussdiagrammen, visuelle Hervorhebung von Code-Fragmenten bspw. beim 'Steppen'[21] durch ein Programm bis hin zu animierten Filmen [vgl. Myers 1986, 60ff; Domingue & Mulholland]. Die besonders Mitte der 1980er und 1990er Jahre rege Forschung in diesem Bereich hat einige Systeme für Anfänger hervorgebracht [siehe Deek & McHugh 1998]. Ein neueres System, welches graphisches Debuggen mit kollaborativem Lernen vereint, ist **ISVL** (Abbildung 2) [Domingue & Mulholland]. Über eine Client-Server-Architektur können Prolog-Programme als Text entwickelt und visuell ausgeführt werden sowie solche Animationen mit Annotationen versehen als Film aufgezeichnet und an andere übermittelt werden.

---

[20] Syntax-Highlighting bezeichnet die visuelle Hervorhebung von Schlüsselwörtern, Strings und Kommentaren zur besseren Strukturierung eines Dokumentes.

[21] Beim Steppen durch ein Programm wird jede Anweisung schrittweise ausgeführt und Veränderungen bspw. in Variablenzuständen so sichtbar gemacht.

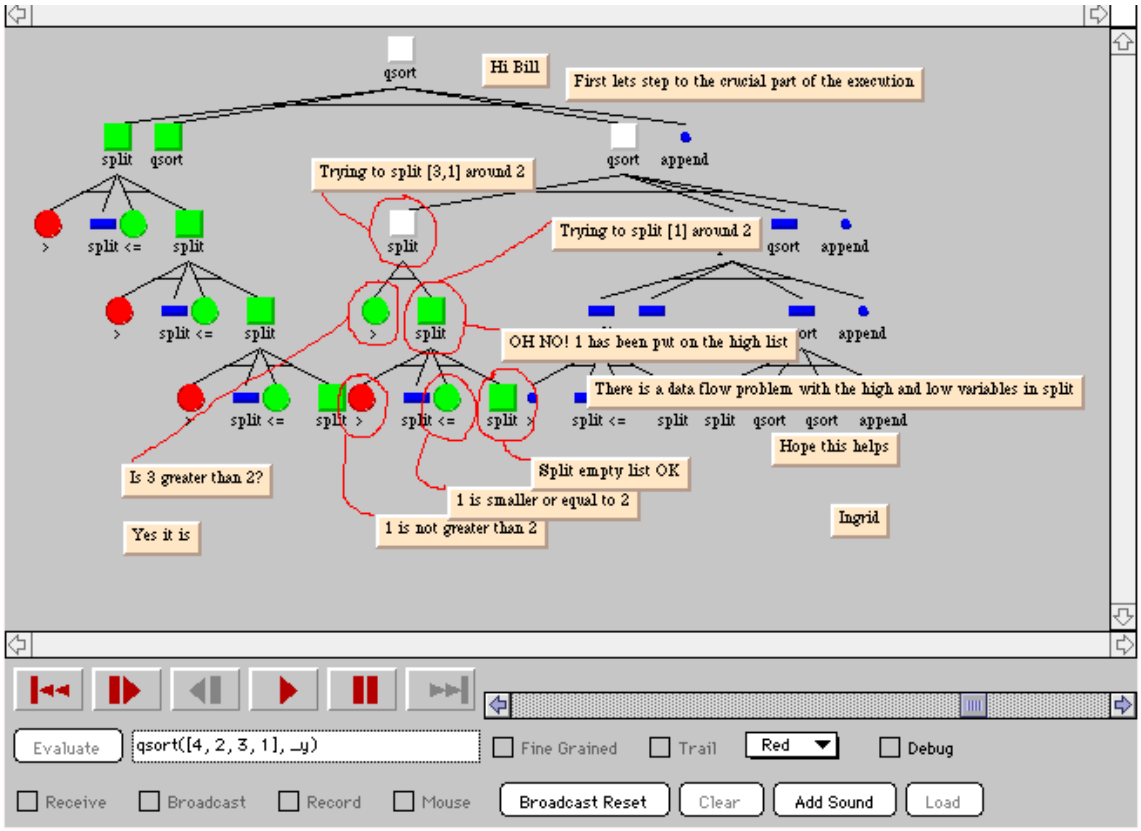

Abbildung 2. Animierte Programmausführung in ISVL. Aus: Domingue & Mulholland

Bisher konnte empirisch jedoch nicht gezeigt werden, dass SV ein besseres Verständnis für OO-Konzepte fördert [vgl. Ben-Ari 2001]. Auch der visuellen Textstrukturierung durch 'Indenting' und andere 'Pretty Printing'-Techniken sprechen bislang nur wenige Studien einen größeren positiven Effekt zu [vgl. Green 1990b, 129ff]. Vielmehr sind es einzelne Aufgaben und individuelle Unterschiede, welche die Nützlichkeit der Darstellungsform stark beeinflussen [vgl. Green 1990b, 127f; Domingue & Mulholland]. Zwar kann auch für Anfänger ein Flussdiagramm den Kontrollfluss nachvollziehbarer und als Prinzip verständlicher machen, in effektiver Weise jedoch scheinbar gerade bei den Fortgeschritteneren [vgl. Domingue & Mulholland]. Als Argument, dass Visualisierungen allein nicht ausreichen können, nennen Domingue & Mulholland zudem eine Tendenz zur falschen Interpretation von Visualisierungen. Durch selektive Wahrnehmung bspw. ziehen Anfänger häufig die Informationen aus der Visualisierung, die ihre bestehenden (fehlerhaften) Annahmen und Erwartungen über das Programm bestätigen. Allerdings bieten sich gerade Anfängerprogramme für solche Werkzeuge insofern an, als dass ihre Größe und Komplexität eine abstrakte graphische, aber überschaubare Visualisierung erlaubt [vgl. Cleve & Zeller 2001, 51ff].

Um Syntaxschwierigkeiten zu mildern und damit die Aufmerksamkeit auf Planung und Semantik eines Programms zu lenken, werden in vielen Anfängerplattformen Formen des visuellen Programmierens bzw. strukturierte Editoren eingesetzt. Hierbei wird nicht nur frei als Text kodiert, sondern Sprachkonstrukte werden als Pseudo-Code über Piktogramme oder per Menü ausgewählt und zusammengestellt. Neal [1989, 63f] argumentiert, dass die Auswahl von Templates über Pseudo-Code-Bezeichnungen oder durch Navigation über Baumstrukturen ein höheres Niveau an Verständnis von Sprachkonstrukten voraussetzt als realiter vorhanden. Insofern Informationen zu Evaluation vorliegen, deuten sie jedoch generell auf die Nützlichkeit des Ansatzes [Green 1990b, 128; Deek & McHugh 1998, 157]. Green [1990b, 128] teilt die Einschätzung, dass die positiven Ergebnisse zur Programmierung mit Pseudo-Code vermutlich auf die geringere Distanz zur (formellen oder informellen) Spezifikation zurückzuführen ist.

BlueJ [Kölling et al. 2003] ist eine Open-Source-IDE für Java-Anfänger, die visuelles Programmieren anhand eines UML-Klassendiagramms unterstützt (Abbildung 3). Konzipiert ist BlueJ als ein einfach zu bedienendes System, dass zur Lehre Objektorientierter Prinzipien eingesetzt werden soll. Entsprechend dem 'objects-first'-Ansatz wird das abstrakte Modellieren von und Denken in Objekten den konkreten prozeduralen Konzepten wie Schleifen, Datentypen etc. vorgezogen [vgl. Balzert & Balzert 2004, 23f]. Klassen werden graphisch angelegt, zugehörige Instanzen mit ihren Parametern über einen Dialog erzeugt und Methoden für ein Objekt können über Menü aufgerufen werden. Die eigentliche Kodierung erfolgt jedoch wiederum textuell mittels Editor. Neben 'Syntax-Highlighting' und Markierung der Zeilen mit Syntaxfehlern hilft der BlueJ-Compiler durch verständlichere und ausführlichere Fehlermeldungen – es wird jeweils nur der erste Fehler herangezogen – als sie der JDK-Standard-Compiler meldet [Lewis & Watkins 2001]. Im Falle einer falsch geschriebenen Variablen bspw. wird zusätzlich zur Standardmeldung '`cannot resolve symbol`' der Hinweis auf mögliche Ursachen gegeben – Typo, fehlende Deklaration einer Variablen/ Methode/ Klasse bzw. ein Fehler in deren Reichweite.

Darüber hinaus integriert BlueJ neben einem Debugger[22] und JUnit-Tests (siehe Kap. 3.2) ein Werkzeug zur interaktiven Inspektion, in dem nach Ausführen einer Methode

[22] ***Debugger*** ermöglichen es, ohne den Code durch das Einfügen zahlreicher Ausgabeanweisungen zu ändern und damit u. U. neue Fehler zu erzeugen, den Programmablauf mit Werten und Wertänderungen nachzuverfolgen. Damit stellen sie ein nützliches Werkzeug zum Prüfen von Hypothesen über Fehler und ihre Ursachen dar, bedeuten für Programmieranfänger aber auch einen zusätzlichen Lernaufwand [adebug, 13; vgl. Allen et al. 2001].

für ein Objekt die Zustandsänderungen seiner Variablen in einem Fenster angezeigt werden. Zwei als Umfragen gestaltete Evaluierungsstudien [Hagan & Markham, 2000; Van Haaster & Hagan 2004] ergaben, dass die Studenten BlueJ als hilfreiche Programmierumgebung einschätzen. Als besonders hilfreich wurden dabei die Hinweise zu den Compiler-Meldungen empfunden. Kölling et al. [2003, 263] berichten, dass die Prüfungs- und Hausaufgabenleistungen ein gutes Verständnis von Objektorientierung bestätigen, jedoch ist kein systematischer Vergleich zu einer Kontrollgruppe ohne das BlueJ-System vorgenommen worden. „The need to leave BlueJ“ bedingt jedoch nach Ansicht der Entwickler Kölling et al. [2003, 264] seine spezifischen Probleme.

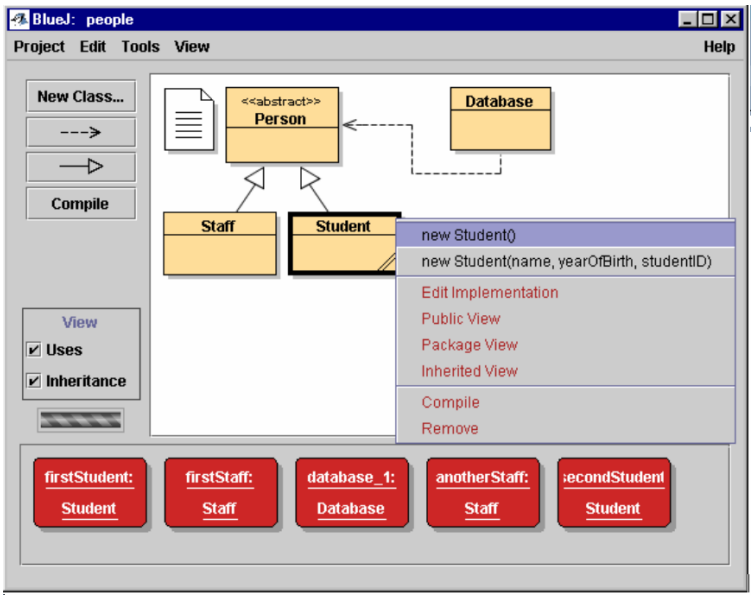

Abbildung 3. BlueJ's Hauptfenster. Aus: Lewis & Watkins 2001

Da Neal [1989, 63f] die Hilfestellung in Form von strukturierten Editoren als unzureichend ansieht, hat sie als zusätzliche Unterstützung mit der Integration einer Beispielbibliothek in die Programmierumgebung einen neuen Ansatz des 'example-based programming' vorgestellt [vgl. Deek & McHugh 1998, 142]. Neben dem Ziel der Veranschaulichung von Sprachkonstrukten an konkreten Beispielen, nennt die Autorin die Bedeutung des Lernens anhand von Beispielen sowie die Wiederverwendung von Software in der Praxis als weitere Motivation für ihr System. Das System umfasst einen strukturierten Editor, in dem über eine Palette Ausdrücke gewählt werden können und Kommandos nach Tastatureingabe vervollständigt werden, sowie ein Fenster für die Anzeige von Beispielen. Diese Beispiele werden über ein Dialogfenster mit den Namen der in ihnen implementierten Funktionen selektiert und können bearbeitet und kopiert, jedoch nicht in die Bibliothek aufgenommen werden.

In einer Evaluierung haben nur sechs der 22 Versuchspersonen die Beispielfunktion gar nicht genutzt, die Verbleibenden gebrauchten sie hauptsächlich während der Codierung zur Hilfe bei Syntax- sowie weniger bei Semantikproblemen und ließen sich im Schnitt ein oder zwei Beispiele zeigen. Als Inspiration für das algorithmische Design und das Finden eines Lösungsansatzes diente die Beispielbibliothek lediglich in geringerem Maße. Das Feedback schildert Neal als überaus positiv, da die Beispiele generell als sehr hilfreich angesehen wurden. Die Autorin folgert daraus, dass der Ansatz viel versprechend ist – vor allem bei Syntax- und Semantikproblemen. Das System könnte aber dahingehend weiterentwickelt werden, dass Beispiele automatisch bspw. anhand semantischer oder struktureller Ähnlichkeiten abgerufen werden.

Während die vorgestellten Systeme und Ansätze Werkzeuge darstellen, das Implementieren und Debuggen eines Programms zu erleichtern, ist es ihnen nicht möglich, Fehler mit ihren Ursachen zu diagnostizieren. Automatische Debugger hingegen verfolgen das Aufdecken von Fehlern, um die Güte eines Programms zu bewerten und Feedback an den Lernenden zu geben.

3.2 AUTOMATISCHE DEBUGGER

Da im Prozess der Softwareentwicklung das systematische Testen auf Fehler bzw. Fehlerfreiheit und Debuggen einen zentralen Stellenwert einnimmt, sind neben den beschriebenen Werkzeugen zur Selbsthilfe viele automatisierte Analyseverfahren entwickelt worden. Generell lassen sich statische Techniken, in denen der Quelltext ohne Ausführung analysiert wird, von dynamischen unterscheiden, die Daten aus dem Laufzeitverhalten eines Programms für konkrete Eingabewerte ziehen [vgl. Balzert 1999, 509][23]. Eine Art der Unterstützung ist das Automatisieren von Testfällen (z. B. Open-Source-Tools JUnit oder DejaGnu), so dass die sukzessive Prüfung der Programme auf für die Spezifikation kritische Eingabe-/ Ausgabepaare und Protokollierung der Ergebnisse abgenommen werden können[24]. Während Testen immer nur exemplarisch die korrekte Funktionsweise für die getesteten Fälle, nicht jedoch eine Freiheit von Fehlern beweisen kann [vgl. Balzert 1999, 533], ist die statische Analyse aufgrund der Fähigkeit zur Abstraktion weniger abhängig von der Auswahl geeigneter Testfälle [vgl. Hovemeyer & Pugh 2004, 93]. Prinzipielles Ziel ist der formale Beweis von Korrektheit

[23] Einen Einführung in intellektuelle Analysemethoden wie 'Review', 'Walkthrough' oder 'Inspektion' bietet u. a. Balzert 1999, 539ff.

[24] Einen Überblick über Techniken des Testens gibt Balzert 1999.

für alle möglichen Eingabedaten (Verifikation)[25], was in der Praxis jedoch in der Regel so komplex ist, dass partielle Techniken eingesetzt werden [vgl. Hovemeyer & Pugh 2004, 93].

Tools wie **ESC/Java** [Flanagan et al. 2002] oder **Bandera** [Corbett et al. 2000], die partielle Verifikation für bestimmte Eigenschaften anstreben [vgl. Hovemeyer & Pugh 2004, 93], erfordern zur erfolgreichen Überprüfung vom Programmierer die Angabe von einzuhaltenden Bedingungen der Spezifikation. **FindBugs** [Hovemeyer & Pugh 2004], **PMD** [PMD] und **JLint** [JLint] hingegen weisen anhand der Syntax und ggf. Informationen aus dem Datenfluss 'Bug Patterns', typische fehlerverdächtige Code-Stellen, oder Verstöße gegen Kodierkonventionen nach [vgl. Almazan et al. 2004, 245ff; Hovemeyer & Pugh 2004, 92ff][26].

Da diese Systeme auf große Softwareprojekte mit komplexen Programmen ausgerichtet sind, testen sie schwerpunktmäßig auf Probleme mit Threads, Sicherheit, Speicherressourcen und Performanz und setzen syntaktische Korrektheit des Programms voraus. Auch wenn es bspw. in PMD möglich ist, eigene Detektoren für spezielle Anfängerfehler zu erstellen, müsste die Eignung für die Unterstützung von Anfängern erst untersucht werden. Hovemeyer et al. [2005] beschreiben die Möglichkeit der Integration einiger Anfängerfehler in den Fehlerkatalog von FindBugs. Ebenso wie ein von Brun & Ernst [2004] vorgestelltes dynamisches Verfahren, bei dem Techniken des maschinellen Lernens zur Identifizierung kritischer Programmeigenschaften eingesetzt werden, leiden diese Tools unter einer hohen Rate (FindBugs bspw. bis zu 50 %) an 'false positives', Warnungen über korrekten Code [vgl. Almazan et al. 2004]. Damit sind sie nicht ohne weiteres für den didaktischen Einsatz geeignet [vgl. Qui 2004, 2]. Im Folgenden werden daher solche statische oder dynamische Analyseverfahren genauer erläutert, die Eingang in Systeme für Anfänger gefunden haben.

3.2.1 Automatisches Debuggen von Syntaxfehlern

Hristova et al. [2003] unterstützen mit **Expresso** Java-Anfänger in ihren ersten Wochen, indem sie durch statische Analyse die in ihrer Umfrage ermittelten 20 häufigsten Anfängerfehler (siehe Kapitel 2.3) in Programmen entdecken und erklärende Korrekturhinweise im Sinne von erweiterten Compiler-Meldungen generieren. Bei den abge-

[25] Eine intuitive Einführung in Verifikation findet sich bei Balzert 1999.

[26] FindBugs, PMD und JLint sind Open-Source-Tools für Java erhältlich unter ihrer Projektseite bei Sourceforge. http://{name}.sourceforge.net

deckten Fehlern handelt es sich fast ausschließlich um jene, die auch ein JDK-Standard-compiler ausgeben würde. Dies sind syntaktische Fehler wie '==' vs. '=', '&&' vs. '&' und Fehler, die Methoden, Parameter und Rückgaben betreffen. Diese sind nach Ansicht der Autoren neben ihrer einfachen Implementierung relevant für die ersten Übungen und allein durch die Compiler-Meldung schlecht nachvollziehbar (Bsp. für '==' vs. '=' : `incompatible types found int required boolean`).

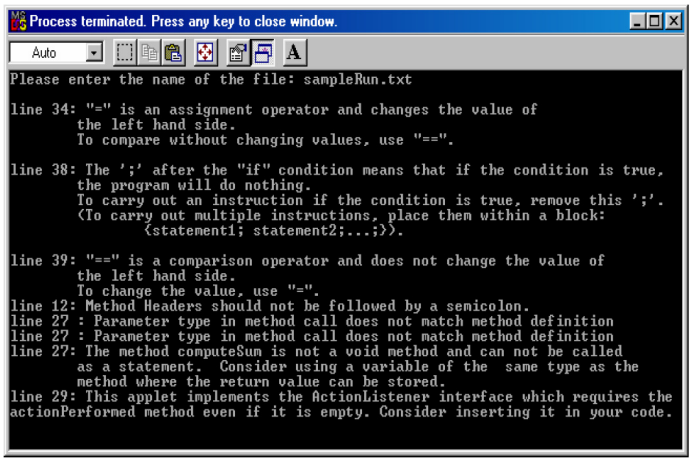

Abbildung 4. Fehlermeldungen in Expresso. Aus: Hristova et al. 2003, 156

Die Implementierung erfolgte als ein mehrstufiger Pre-Prozessor geschrieben in C++. Kommentare und Zwischenräume werden aus der Programmdatei gelöscht, der restliche Text in Tokens zerlegt und die Strings mit den Fehlermustern abgeglichen [vgl. auch Bancroft et al. 2004]. Eine Evaluierung steht noch aus. Interessant wäre besonders, wie lange eine solche Unterstützung als hilfreich empfunden wird und welche Auswirkungen auf das Verständnis 'normaler' Compiler-Meldungen festzustellen sind.

Einen elaborierten Ansatz zum automatischen Debuggen von Syntaxfehlern bei eingeschränktem Sprachumfang – nur Variablen, Operatoren und Schleifen – stellen Franek & Sykes [2004] vor. Liegt eine Musterlösung für eine Aufgabe vor, wird der nicht kompilierbare Java-Code des Lernenden[27] so transformiert, dass er auf den String der Lösung abgebildet werden kann. Da dieses Pattern-Matching nur bei vorhandener Referenzlösung durchführbar ist, dann aber keine Varianten wie bspw. eine 'for'- statt einer 'while'-Schleife zulässt, schlagen die Autoren zusätzlich ein Modul zur Erkennung der Intention des Programmierers vor. Ein Scanner untersucht dabei den Code des Lernenden und versucht, ein bekanntes Token zu extrahieren. Hierzu werden Vergleiche mit

[27] Eine Aufgabe besteht hier aus dem Vervollständigen eines Code-Gerüsts bspw. durch eine Schleife.

Listen über alle in Java reservierten Wörter und Schlüsselwörter sowie über Operatoren und im Programm bereits verwendete Variablen- bzw. Methodennamen vorgenommen. Ist das aktuelle Token auf diese Weise nicht zuzuordnen, werden solange korrigierende Transformationen eingesetzt bis ein Token erzeugt ist, welches der Parser akzeptiert. Das korrigierte Token kann damit nicht irgendein beliebiges gültiges Symbol sein, sondern muss in die derzeit vom Parser betrachtete grammatikalische Regel einfügbar sein. Ist der vorangegangene Code bspw. als Beginn einer 'for'-Schleife vom Parser erkannt, erwartet dieser nun ein Token, mit dem die Grammatik für 'for' ('init', 'test', 'increment') erfüllt wird. Die Transformation dient damit als ein 'Repair', damit der Parsing-Prozess fortgesetzt werden kann, wo der Standard-Java-Parser (Compiler) aufgrund von unauflösbaren Mehrdeutigkeiten im Hinblick auf anzuwendende Produktionen einen Fehler melden und abbrechen würde[28].

Zur eindeutigen Zuordnung des für den Konflikt und folglich den Abbruch des Parsers ursächlichen Fehlers, untersuchen Fox & Uti [2003] den Einsatz maschineller Klassifikationsverfahren. Für vier Compiler-Fehler zeigen sie die möglichen Ursachen auf – so kann die Meldung `")" expected` sieben verschiedene Gründe haben, von denen nur zwei tatsächlich einem Klammerproblem entsprechen. Jedem der vier Compiler-Fehler kann so eine Fehlerhierarchie mit Fehlerklassen zugeordnet werden. Wird der Syntaxfehler ') ' vom Compiler ausgegeben, erfolgt eine automatische Klassifizierung in eine der Kategorien, indem für eine Kategorie aussagekräftige Tests durchgeführt werden.

Das Ermitteln, ob eine falsch geschriebene Variable den Fehler herbeigeführt hat, wird wiederum mithilfe einer Distanzberechnung (Levenstheins 'Edit Distance') zwischen Strings realisiert. Wird in der Fehlerzeile ein unbekanntes Token (kein Schlüsselwort, Methoden- oder Variablenname, Operator etc.) gefunden, wird der 'closest match' zu den bekannten Worten durch nicht weiter erläuterte Abbildungsverfahren gesucht. Im Gegensatz zum Verfahren von Franek & Sykes [2004] ist hier das explizite Modellieren von Fehlerursachen und deren Syntaxmustern nötig, während bei Franek & Sykes Tokens – eingegrenzt durch die Anforderung auf grammatikalische Plausibilität – flexibler korrigiert werden können. Ob der scheinbar höhere Modellierungsaufwand durch höhere Genauigkeit im Einklang mit der Intention des Programmierers zurechtfertigen ist, ist in Folge fehlender Evaluierung beider Systeme nicht abzuschätzen.

---

[28] Eine kurze Einführung in die Arbeitsweise eines Parsers geben bspw. Krinke & Zeller [2004]. Zu ‚error recovery' in Parsern auch Cerecke [2001]

Im Gegensatz zu den bisher beschriebenen Techniken zum Debuggen von Syntax konzentrieren sich die meisten Systeme zur Unterstützung der Anfänger auf die Entdeckung semantischer Fehler bzw. Abweichungen von der Problemspezifikation.

3.2.2 Automatisches Debuggen von Semantischen und Spezifikationsfehlern

Die Analysekomponente von **ELP** [Bancroft et al. 2004; 2005] umfasst neben Wissen über Programmierprinzipien und Strukturen der Sprache vor allem Wissen über die Aufgabenstellung. Mit dem Ziel, ein automatisches Benotungssystem für Lösungen zu entwickeln und gleichzeitig schnelles Feedback zur Ausbildung adäquater mentaler Modelle zu ermöglichen, werden 'Fill-in-the-Gap'-Aufgaben in Java sowohl statisch als auch dynamisch getestet. Ist der Lösungsversuch vom Compiler übersetzbar, wird in der statischen Analyse der Code zuerst mithilfe eines ANTLR-Parser[29] in einen Abstrakten Syntax Baum (AST) in XML-Format überführt. Auf Basis dieser AST-Repräsentation des Codes werden in Anlehnung an Softwaremetriken Code-Statistiken wie Anzahl der Variablen, Anweisungen, logische Entscheidungen ('cyclomatic complexity') oder die Zeilenlänge erhoben und auf Fehler wie ein vergessenes 'break' oder 'default' in einer 'switch'-Anweisung geprüft. Um konkretes Feedback im Hinblick auf die Problemstellung liefern zu können, wird zudem dieser AST mit dem AST einer oder mehrerer Musterlösungen verglichen. Damit zulässige Variationen erkannt werden, werden beide ASTs normalisiert, so dass bspw. der generische Knoten 'loop' für alle Schleifentypen steht und eine 'while'-Schleife in der Studentenlösung als äquivalent zu einer 'for'-Schleife der Musterlösung behandelt wird.

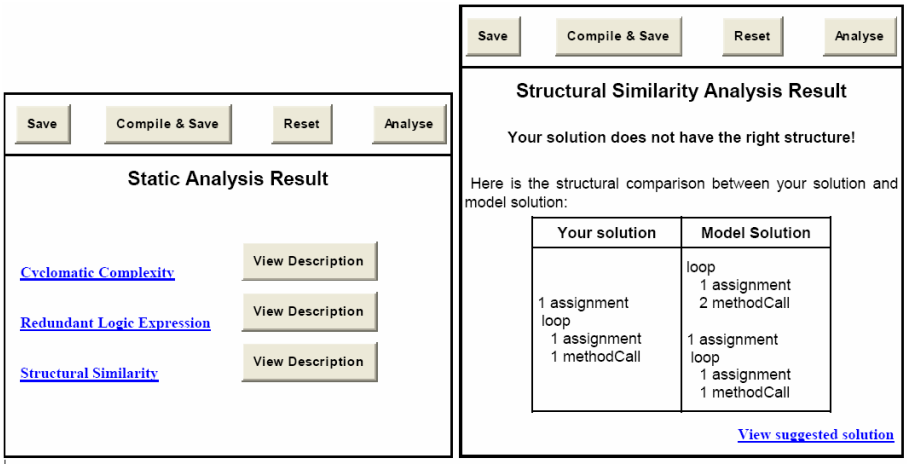

Abbildung 5. Auswertung der statischen Analyse in ELP. Aus: Bancroft et al. 2004, 323f

[29] ANTLR ist ein Parsergenerator, mit dem ein Parser zu einer spezifizierten Grammatik bspw. für die Notation von Java erstellt werden kann [vgl. Krinke & Zeller 2004, 111-121]. Weitere Information sowie kostenloser Download unter www.antlr.org.

Positives sowie negatives Feedback wird sowohl zu den ausgewerteten Metriken als auch zu den Ergebnissen des strukturellen Vergleichs zur Referenzlösung gegeben. Zudem kann der Code der Musterlösung angezeigt werden.

Um darüber hinaus Fehler in der Funktionalität und mögliche Ursachen rückmelden zu können, wurde das Analyse-Framework um dynamische Black-Box- und White-Box-Tests erweitert. Im ersten Fall werden für kritische Eingabewerte, die als Testfälle zu den jeweiligen Aufgaben bereitgestellt werden müssen, die Ausgaben des Lösungsversuchs – nach Extraktion von Schlüsselworten und Normalisierung bzgl. Tabs oder Leerzeichen, um irrelevante Unterschiede ignorieren zu können – zur Musterlösung verglichen.

Schlagen ein oder mehrere Tests fehl, wird im 'White-Box'-Test die verantwortliche Stelle des Codes genauer untersucht. Dazu wird der Code des Lernenden ebenso wie die entsprechende Lücke der Musterlösung in speziell zu einer Aufgabe bereitgestellten Adapter-Code eingebettet und ihre Eingabe-/ Ausgabewertpaare gegenübergestellt. Für die Fehlerlokalisierung muss der Adapter-Code je nach Aufgabe Mechanismen zum Nachvollziehen der Vorgänge innerhalb der Lücke stellen: Veränderungen des Speicherzustands von Variablen, Zusicherung für einzuhaltende Werte ('Assertions') oder Ausgabeanweisungen.

Als Feedback für den dynamischen Teil der Auswertung wird zurückgemeldet, welche Tests erfolgreich oder nicht erfolgreich waren und im Fehlerfall potentielle Gründe benannt, welche vorher zu den Testfällen als Annotationen manuell zu erstellen sind. Wie verständlich und hilfreich diese Auswertungen von Anfängern empfunden werden, kann leider nicht erörtert werden, da eine Evaluierung an Studierenden noch aussteht. Einige Nachteile sind jedoch deutlich. Statische und dynamische Analyse können nur auf Programmen frei von Syntaxfehlern arbeiten, so dass vorher keine Hilfestellung möglich ist. Spezifisch in ELP ist, dass zur Einschränkung der Variationsbreite bei der statischen Strukturanalyse zudem nur wohlgeformte Lücken (eine Anweisung, ein Block, eine Klasse) analysiert werden können [vgl. Bancroft et al 2004]. Die dynamischen Tests sind bisher nicht zur Auswertung von Aufgaben zur GUI-Entwicklung und zu Dateiein- bzw. ausgaben einsetzbar. Der anfängliche Aufwand für die manuelle Erstellung von Testfällen und den so entscheidenden Hinweisen für ein Fehlschlagen sind nicht zu unterschätzen [vgl. Krinke et al. 2001, 10].

Ähnlich dieser ELP-Architektur sind einige Systeme im universitären Einsatz angelegt. Der im Würzburger Java Online Praktikum (**JOP**) integrierte **Praktomat** der Universität Passau prüft Aufgaben auf Kompilierbarkeit und bei Erfolg mithilfe des Open-Source-Tools DejaGnu[30] gegen Testfälle und über Checkstyle[31] auf Code-Konventionen [JOP-Praktomat; Krinke et al. 2002; 2001]. Krinke et al. [2001] berichten, dass in einer Umfrage 57 % der Studenten die automatisierten Tests als effektiv ansehen und für die Dozenten Korrektur- und Bewertungsaufwand stark gesunken ist. Jedoch ist die Erstellung von ausreichend geeigneten Testfällen zu jeder Aufgabe sehr kostenintensiv [vgl. Krinke et al. 2001, 10] und damit wird die Flexibilität in Bezug auf die Einführung neuer Aufgabenstellungen praktisch eingeschränkt. Praktomat gibt im Gegensatz zu ELP keine erklärenden Hinweise, warum die Testfälle nicht bestanden worden sind, sondern stellt nur erwartete Werte den tatsächlichen gegenüber. Dieses alleinige Rückmelden von Testergebnissen sehen Johnson & Soloway [1984, 369] als unzureichend an, da dies zwar zeigt, dass ein Fehler im Programm vorhanden ist, die Ursache aber nicht festgestellt wird. Zudem können oft mehrere Fehler die gleiche Ausgabe erzeugen [Johnson & Soloway 1984, 369].

Auch in **Projectory** [Gößner et al. a, b], einem Projekt im Rahmen des E-Learning Network Niedersachsen (ELAN)[32], werden die gemäß dem Ansatz des 'Pair Programming' in Zweierteams entwickelten Aufgaben bei Abnahme über JUnit-Tests automatisch getestet. Eine Befragung unter Studenten ergab hier eine große Wertschätzung, stuften doch ca. 70 % die automatisierten Tests auf einer sechsstufigen Skala mit Note eins oder zwei als (sehr) sinnvoll ein.

Das automatische Debugging-System **Laura** [Adam & Laurent 1980] gründet seine Fehlererkennung hingegen allein auf statische Analysetechniken. Syntaktisch korrekte Anfängerprogramme in FORTRAN werden in einen von der Sprachnotation losgelösten Graphen überführt und zum Graphen der Referenzlösung verglichen. Dabei wird versucht, die Gleichwertigkeit beider Programme zu beweisen, indem mithilfe diverser Heuristiken der aktuelle Graph in den der Modelllösung transformiert wird [vgl. Ramadhan & Shihab 2000, 162]. Ein Beispiel hierfür ist das Anlegen einer neuen Varia-

---

[30] DejaGnu ist ein freies Framework für automatisiertes Testen. http://www.gnu.org/software/dejagnu/ Einführung in die Nutzung geben Krinke & Zeller [2004].

[31] Checkstyle als Open-Source-Tool ermöglicht die Automatische Prüfung auf Einhaltung selbst konfigurierbarer Kodierrichtlinien. http://checkstyle.sourceforge.net/ . Einführung in die Nutzung geben Krinke & Zeller [2004].

[32] Projectory in ELAN siehe http://www.l3s.de/elan/kb3/index.php?id=574 ; unabhängige Projektseite: www.projectory.org

blen, wenn eine Variable im Anfängerprogramm an mehreren Stellen quasi als verschiedene unabhängige Variable genutzt wird [nach Ramadhan & Shihab 2000, 162]. Das Wissen des Systems über solche zulässigen Normalisierungen bzw. Transformationen ermöglicht den Abgleich der einzelnen Lösungsschritte des Anfängers zu den Teilen der Musterlösung [vgl. Ramadhan & Shihab 2000, 162]. Alle ermittelten Abweichungen werden als Fehler zurückgemeldet, da angenommen wird, dass die Ziele der Programmspezifikation nicht erfüllt werden.

Laut Deek & McHugh [1998, 145] ergab eine Evaluation an 100 Programmen zu acht Problemstellungen zufriedenstellende Ergebnisse, da trotz Variationen in der Struktur korrekte Programme erkannt sowie Fehler in den anderen gefunden werden konnten. Problematisch bei diesem Ansatz ist jedoch, dass stark abweichende, aber zulässige Variationen in einem Programm das System unbrauchbar machen [vgl. Ducassé & Emde 1988, 165]. Zudem wird eine zuverlässige Diagnose dadurch erschwert, dass erst komplette Lösungsversuche analysiert werden, die dann mehrere miteinander interagierende Fehler aufweisen können [vgl. Ramadhan & Shihab 2000, 161]. Ähnliche Schwierigkeiten treten auch bei dem automatischen LISP-Debugger **Talus** auf, auch wenn hier mithilfe formaler Verifikationstechniken die Äquivalenz zu mehreren Referenzalgorithmen zu beweisen versucht wird [vgl. Ducassé & Emde 1988, 165; Ramadhan & Shihab 2000, 163f].

Die Berechung der Gleichwertigkeit zu einer Referenzlösung oder mehreren Algorithmen ist nach Ducassé & Emde [1988] der einfachste, aber schwächste Weg der Diagnose. Ebenso wie beim Debuggen spezieller Aspekte z. B. Syntax und beim Prüfen auf Testfälle wird oft nur eine Teilmenge möglicher Fehler erfasst [vgl. Johnson & Soloway 1984, 369]. Zur Überwindung dieser Problematiken modellieren ITS Domainwissen aufgabenunabhängig.

3.3 INTELLIGENTE TUTORIELLE SYSTEME

ITS zeichnen sich dadurch aus, dass sie mithilfe von Wissen über das jeweilige Fachgebiet, den Lernenden und allgemeine Lehrstrategien eine Individualisierung des Lernprozesses anstreben [Brusilovsky 1999, 19f]. Im Gegensatz zum behavioristischen Paradigma wird der Lernende nicht mehr als eine 'black box' betrachtet, sondern gemäß kognitivistischen Theorien stehen seine individuellen kognitiven Prozesse bei der Informationsverarbeitung und -speicherung im Mittelpunkt [vgl. Zielhofer 2003, 9]. So „soll ein

ITS die Individualisierung […] erreichen, indem es den Lernenden analysiert und seinen Tutor an den jeweiligen Stand «intelligent» anpassen kann“ [Schulmeister 1997, 189].

Brusilovsky & Peylo [2003, 158] unterscheidet dabei die Techniken von ITS gemäß ihrem Schwerpunkt in 'curriculum sequencing', 'intelligent solution analysis' und 'interactive problem solving support'. Im 'curriculum sequencing' wird versucht, für den Lernenden den optimalen Pfad durch das Lehrmaterial zu ermitteln, sei es aktiv hinführend zu einem vorgegebenen Lernziel oder passiv zur Überwindung festgestellter Wissenslücken bzw. fehlerhaften Wissens [vgl. Brusilovsky 1999, 19f]. Im Bereich der Programmierung dominieren hingegen ITS zur intelligenten Analyse von Lösungen und interaktiver Problemlöseunterstützung. Intelligente Analyse meint dabei, dass für falsche Antworten bzw. fehlerhafte Lösungsversuche genau bestimmt werden soll, welcher konkrete Fehler wo begangen wurde und welches Wissen dem Lernenden damit fehlt oder inkorrekt ist [vgl. Brusilovsky & Peylo 2003, 159].

Eine wichtige Komponente eines ITS zur Erreichung dieser Ziele ist das Experten- bzw. Wissensmodell, welches die Wissensbasis (deklarativ und prozedural) eines Bereichs abbildet. Diese wird zumeist definiert als Begriffe mit ihren Attributen, Relationen der Begriffe zueinander, z. B. in Form eines semantischen Netzes, sowie Regeln zur Anwendung dieser Konzepte [vgl. Schulmeister 1997, 182f]. Das Expertenmodell kann darüber hinaus allgemeine Heuristiken, die unabhängig von einem konkreten Inhalt als quasi domänenunabhängige Operatoren beim Problemlösen einsetzbar sind – „Finde alle Unbekannten in …“ [Schulmeister 1997, 182f] – umfassen. „Der Inhalt der Wissensbasis muss jedenfalls geeignet sein, Schlussfolgerungen ziehen und Probleme lösen zu können“ [Schulmeister 1997, 183]. „This knowledge base is used to evaluate student progress toward the solution, and to compare and verify results to form the basis for error detection and correction” [Deek & McHugh 1998, 148].

Den Fortschritt im Lernprozess, das aktuelle Wissen eines Lernenden, erfasst das Lerner- bzw. Diagnosemodell eines ITS. Dabei wird im 'subset model' oder auch 'overlay model' das Wissen durch 'Abhaken' als Teilmenge des Expertenwissens repräsentiert, um daraufhin den optimalen Pfad der Präsentation von Inhalten oder Aufgaben individuell zu bestimmen. Im 'deviation model' bzw. 'buggy model' hingegen wird auf das (inkorrekte) Wissen eines Lernenden aus den Abweichungen seiner Antworten zu „der Performanz eines Experten in denselben Situationen“ [Schulmeister 1997, 183] ge-

schlossen [vgl. Schulmeister 1997, 183f]. Dies erfordert eine Erweiterung der Wissensbasis um Fehlerbibliotheken, damit in Hinblick auf eine Fehlerkorrektur durch den Tutor diese Abweichungen spezifiziert werden können und nicht bei Differenzen allgemein fehlendes Wissen angenommen wird [vgl. Schulmeister 1997, 184f].

Die Sammlung und Modellierung typischer Fehler bedeutet einen immensen Entwicklungsaufwand, vor allem, wenn in einem Bereich Fehlermuster nicht über längere Zeit stabil sind oder neue hinzutreten können [vgl. Schulmeister 1997, 185]. Zudem treten gerade bei Anfängern Fehler oft unsystematisch oder im Sinne der 'merged plans' als Mischformen auf, so dass die vorherzusehenden Fälle und damit der Suchraum erheblich anwächst [vgl. Schulmeister 1997, 185]. Aus diesem Grund werden auch Techniken des maschinellen Lernens eingesetzt, um aus Abweichungen zum korrekten Expertenwissen Fehlerbibliotheken automatisch aufbauen oder erweitern zu können [vgl. Shimura & Sison 1998; Numao et al. 2000; Cruz & Sison 2002]. Begangene Fehler oder vermutlich fehlendes Wissen werden im Lernermodell festgehalten und bilden die Basis für die tutorielle Komponente, „as it determines what to teach, when to teach, and how to teach“ [Deek & McHugh 1998, 148][33]. Wiederum variiert die Entwicklung der genannten Komponenten in verschiedenen ITS oft erheblich hinsichtlich des Umfangs des modellierten Wissens [vgl. Schulmeister 1997, 189f].

3.3.1 PROUST

PROUST [Johnson 1990, auch Johnson & Soloway 1984; Johnson 1986] diagnostiziert Fehler in syntaktisch korrekten Pascal-Programmen, indem es basierend auf der Theorie von Programmzielen und -plänen die Intention des Programmierers zu rekonstruieren versucht. Fehler sind damit Abweichungen von der Problemspezifikation, die sich in einer fehlerhaften oder fehlenden Implementierung nötiger Ziele manifestieren. Dafür ist eine umfangreiche Wissensbasis erforderlich, in der typische Teilaufgaben eines Programms bspw. die Mittelwertberechnung ('Average') und deren verschiedene Implementierungsmöglichkeiten in Form von Programmierplänen aufgabenunabhängig definiert werden. Die Plan-Templates als wiederkehrende Code-Muster sind nach Beispielen in Lehrbüchern sowie aus (Anfänger-)Programmen konstruiert und im Expertenmodell zusammen mit den Zielen als 'Frames' in einem Netzwerk organisiert. Vorraus-

[33] Zum Tutorenmodell und Kommunikationsstrategien siehe Schulmeister [1997,186ff].

setzung für die Analyse ist eine Spezifikation der zu bearbeitenden Problemstellung durch Angabe der Ziele, die eine korrekte und vollständige Lösung enthalten muss.

Von dieser Problembeschreibung ausgehend versucht nun PROUST entsprechend eines 'Analyse-durch-Synthese'-Ansatzes [Weber 1994, 262] nach und nach die einzelnen Ziele im Programm zu finden, indem die möglichen Pläne zur Implementierung des jeweiligen Ziels gegen den Code gematcht werden. Da gerade bei Anfängern aber häufig Fehler und auch Variationen bei der Implementierung von Zielen auftreten, verfügt PROUST zudem über verschiedene Transformationsregeln, um möglichst alle Code-Fragmente interpretieren zu können. Realisiert ein Programm bspw. ein anderes Ziel als in der Spezifikation vorgegeben – feste Anzahl von zu lesenden Eingabewerten statt Lesen bis ein Stoppwert eingegeben – können 'goal reformulations' greifen. Dazu beinhaltet ein Zielframe neben Verweisen zu korrekten Plänen auch Regeln, durch welche Ziele es ersetzt werden kann, da es mit diesen erfahrungsgemäß oft verwechselt wird.

Andere Transformationsregeln sind die 'plan-difference rules', welche die Unterschiede zwischen einem Plan der Wissensbasis und einem speziellen Code-Fragment so erklären sollen, dass es als eine gültige Variante erkannt wird oder der zugrunde liegende Fehler in der Umsetzung des Plans bestimmbar ist. Durch schrittweise Anwendung solcher Regeln, die u. a. typische Fehlermuster kodieren, soll ein Plan so umgeformt werden, dass er gegen den tatsächlichen Code abgeglichen werden kann. Mit welchen Regeln dies erreicht wird, gibt Aufschluss darüber, welche Fehler das Programm enthält bzw. ob eine zulässige Variation vorliegt. Damit die Analyse in einem vertretbaren zeitlichen Rahmen zu bewerkstelligen ist, nutzt PROUST des Weiteren Heuristiken zur effizienten Ziel- und Planselektion, zur Auflösung von Mehrdeutigkeiten zwischen (partiell) matchenden Plänen und zur Eingrenzung anzuwendender Transformationsregeln.

Für die ermittelten Fehler werden nach Abschluss der Analyse ausführliche Fehlermeldungen generiert und in Textform ausgegeben. PROUST besitzt keine weiterführenden Strategien zur Instruktion oder Kommunikation, sondern beschränkt sich auf das intelligente Debuggen von Programmen. Durch das beschriebene Verfahren gelingt es PROUST im Gegensatz zu anderen Analysemethoden, verschiedenste semantische und logische Fehler festzustellen, zu lokalisieren und zu erklären [vgl. Johnson & Soloway 1984, 369]: vergessene Pläne z. B. Initialisierungen, Prüfen auf Ausnahmewerte, fehlerhaft implementierte Ziele oder Anweisungen an einer falscher Stelle. Integriert in einen

Pascal-Compiler werden von PROUST jedoch erst syntaktisch korrekte Programme analysiert [vgl. Weber 1994, 262].

Das System wurde an 206 Programmen von Studenten zur Lösung des 'Rainfall'-Problems[34] evaluiert, von denen 183 (89 %) Fehler enthielten. 167 (81 %) Programme konnten von PROUST komplett analysiert werden, wobei 94 % ihrer Fehler korrekt erkannt wurden. Bei 31 (15 %) Programmen gelang nur eine partielle Analyse, so dass 63 % der darin aufgetretenen Fehler nicht berichtet wurden. Die Ergebnisse müssen aber relativiert werden, denn in der Evaluierung an 64 Anfängerprogrammen zu einer anderen Problemstellung – die nicht Basis für die Entwicklung der Transformations- und Fehlerregeln war – konnten nur 50 % der Programme komplett analysiert und davon 91 % der Fehler ermittelt werden. In den 41 % partiell untersuchten Programme wurden 67 % der Fehler nicht entdeckt [vgl. Johnsson 1990, 91-94].

Um die Performanz gerade in Hinblick auf 'false alarms' (kein oder nicht der tatsächliche Fehler) zu verbessern, schlägt Johnson [1990, 95] den Einsatz einer tutoriellen Komponente vor, die durch Fragen an den Lernenden zu einer angemesseneren Selektion zwischen Alternativen beitragen soll. Die Ergebnisse verdeutlichen PROUSTs Abhängigkeit von einem umfassenden und spezifischen Domainwissen in Form von aufgabenübergreifenden Plänen und Regeln [vgl. Schulmeister 1997, 194]. Der Preis für diese intentionsbasierte Fehlerdiagnose ist hoch, da nicht dokumentierte Fehler oder neue Pläne vom System nicht unterstützt werden [auch Deek & McHugh 1998, 152], die Modellierung und ständige Anpassung der Wissensbasis aber sehr zeitintensiv ist.

Adaptivität ist in PROUST insofern realisiert, dass bei wiederholtem Auftreten desselben Fehlers die Fehlermeldung angepasst wird. Während PROUST erst zur Analyse (fast) vollständiger Lösungsversuche einsetzbar ist, zielt der LISP-Tutor auf aktive Lenkung des Lernenden durch Unterstützung bei den Einzelschritten hin zur Problemlösung. Dadurch gelingt es dem System Mehrdeutigkeiten in Folge multipler Fehler in einem Programm, welche die Analyse in PROUST erschweren, zu umgehen [vgl. Ramadhan & Shihab 2000, 161].

[34] Beim 'Rainfall'-Problem soll für eingegebene Niederschlagswerte der durchschnittliche Niederschlag berechnet werden.

3.3.2 LISP-Tutor

Der **LISP-Tutor** [Anderson & Reiser 1985; Anderson et al. 1990; Anderson & Swarecki 1986], in frühen Versionen findet sich auch die Bezeichnung *GREATERP*, wurde zusammen mit weiteren ITS für andere Domänen entwickelt, um Anderson's ACT-Theorie über den Erwerb komplexer kognitiver Fertigkeiten zu testen. Die Theorie geht im Wesentlichen davon aus, dass derartiges prozedurales Wissen für das Lösen komplexer Probleme als eine Menge von Produktionsregeln in Form von 'If-Then'-Regeln repräsentiert wird[35]. Eine grundsätzliche Annahme ist dabei, dass Feedback unmittelbar nach einem Fehler erfolgen muss, damit keine Frustration, keine Situation langwierigen Probierens falscher Lösungen eintritt und vor allem, damit die fehlerhafte Annahme als Ursache korrigiert werden kann [vgl. Anderson et al. 1990, 38]. Dementsprechend sehen die Autoren eine Lernsituation mit einem privaten Tutor als die bestmögliche an, die sie durch Umsetzung der kognitiven Theorie mit ihrem ITS nachzubilden verfolgten: „It approaches the effectiveness of a human tutor" [Anderson & Reiser 1985, 159].

Um den Lernenden bei jedem Schritt im Problemlöseprozess zu unterstützen, überwacht der Tutor im Prozess des 'model tracing' jede Aktion – ein atomarer Ausdruck oder eine Klammer [vgl. Weber 1994, 258] – „and insists that the student stay on a correct path" [Anderson et al. 1990, 30]. Dieser optimale Pfad, der den idealen Lerner spiegelt, wird mithilfe von auf GRAPES basierenden [vgl. Farrell & Sauer 1982] Produktionsregeln im Expertenmodell ermittelt. Ausgehend von einer Spezifikation der Aufgabenstellung kann das zielorientierte Produktionssystem durch sukzessive Anwendung solcher Produktionsregeln vorgegebene Ziele durch Teilziele und Code-Fragmente ersetzen und letztlich ein LISP-Programm erstellen.

IF	the goal is to code the body of a function involving an integer argument
THEN	try integer recursion and set subgoals to plan a terminating case and a recursive case
	[Anderson & Swarecki 1986, 843]

Solange der Lösungsversuch des Lernenden durch die gleiche Sequenz an Produktionsregeln erzeugt werden kann, wie sie im optimalen Pfad für eine Problemstellung festgelegt sind, bleibt der Tutor im Hintergrund. Kodiert wird dabei strikt 'top-down' von links nach rechts in einem syntax-gesteuerten Editor, der durch Bereitstellung von Platzhaltern/ Templates mit automatischer Klammersetzung Syntaxfehler vermeidet und

[35] Eine ausführlichere Beschreibung der ACT-Theorie findet sich bei Anderson et al. [1990], auch Schulmeister [1997, 119].

somit das Arbeitsgedächtnis entlastet [vgl. Weber 1994, 258]. Lässt sich das aktuelle Symbol bzw. der Ausdruck nur durch eine Produktionsregel, die im Kontext nicht ideal ist, oder durch eine der 'buggy rules' erklären, greift der Tutor berichtigend ein und versucht, den Lernenden zur Lösung zu leiten.

Während einer Sitzung wird aus diesen Informationen über die Anwendung korrekter oder fehlerhafter Regeln ein individuelles Lernermodell konstruiert, das den aktuellen Wissenstand modellieren soll. Zeigt sich darin, dass ein Fehler wiederholt begangen wurde, präsentiert der Tutor mehrere Beispiele zur Durcharbeitung des betreffenden Konzeptes. Zur Unterstützung der Planung des Funktionsdesigns aber auch zur Disambiguierung des interpretierten Lernerverhaltens erfolgt die Kommunikation mithilfe eines Menüs, bei dem aus mehreren Alternativen der nächste Schritt gewählt wird.

Evaluierungen des Tutors ergaben eine effektive Fehlererkennung zwischen 45-80 % je nach LISP-Lektion/Komplexität und eine geringfügige Verbesserung der Leistung der Lernenden in einem 'Papier-und-Bleistift-Test' zu komplexen Problemen [Anderson et al. 1990, 31]. Kritik am LISP-Tutor zielt vor allem auf „(d)ie enge Beobachtung des Lernenden …, das unmittelbare Feedback“ [Schulmeister 1997, 119] und seine hohe Direktivität, die ein selbstgesteuertes entdeckendes Lernen unmöglich machen und stark erinnern an behavioristische Methoden des Programmierten Lernens [vgl. Schulmeister 1997, 119, 213f].

Viele Studien betonen den positiven Einfluss direkten Feedbacks, jedoch konnten Schooler & Anderson [1990] zeigen, dass Lerner in dieser Bedingung größere Schwierigkeiten haben können, Programme systematisch nach Fehlern zu durchsuchen und zu korrigieren. Das sofortige Eingreifen ist jedoch auch eine wichtige Voraussetzung für die Reduzierung des Berechnungsaufwands, da es kombinierte Fehler vermeidet [vgl. Ramadhan & Shihab 2000, 164]. Deek & McHugh [1998, 165f] fügen zudem an, dass die Art der Aufgabenstellung im LISP-Tutor mit ihren expliziten Angaben und das Planen über Menüauswahl – Studenten beklagten die Menüs – die Ausbildung allgemeiner und algorithmischer Problemlösefähigkeiten vernachlässigten.

In einer neueren Version des Systems ist es möglich, die Kodierreihenfolge frei zu wählen und Feedback nur bei Bedarf anzufordern. Die Diagnose beruht dabei weiterhin auf dem umfangreichen Produktionssystem (in der Version von Anderson & Swarecki [1986] insgesamt 1200 'ideale' und 'buggy' Produktionsregeln). Während intelligente Entwicklungsumgebungen die Problematik der Abhängigkeit von einer umfangreichen

Wissensbasis gerade hinsichtlich der 'buggy rules' zur Erklärung von Fehlern teilen, lassen sie dem Lernenden mehr Freiraum zum explorativen Lernen.

3.4 INTELLIGENTE ENTWICKLUNGSUMGEBUNGEN

Als Reaktion auf die Kritik an ITS wie dem LISP-Tutor sind intelligente Entwicklungsumgebungen entwickelt worden, um dem Lernenden durch Bereitstellung von Werkzeugen – Visualisierungen, Beispiele, Debugger – selbstständiges Lernen und Fehlerfinden sowie intelligentes Tutoring in Problemsituationen zu ermöglichen [siehe Ramadhan & Shihab 2000, 158f].

3.4.1 ELM (Episodic Learner Model)

ELM-ART[36] [Brusilovsky & Weber 2001] ist ein umfassendes web-basiertes tutorielles System, dass dem Lernenden neben individuellem Feedback, adaptive Wissensvermittlung mittels 'curriculum sequencing', aber auch ein Steppermodul zur schrittweisen Ausführung und Visualisierung eines Programmbeispiels bietet. Über diese Online-Plattform können Lektionen zu den Konzepten der Programmierung in LISP durchgearbeitet und erworbene Kenntnisse an Multiple-Choice-Fragen und Übungsaufgaben getestet werden. Im Lernermodell wird entsprechend den Antworten das Wissen des Lernenden als Teilmenge eines Experten modelliert und daraufhin Empfehlungen zur weiteren Navigation durch den Lehrstoff unterbreitet. Dies geschieht in Form von Markierungen der Hyperlinks, die angeben, ob der betreffende Lehrstoff bereits bekannt, hinsichtlich des aktuellen Kenntnisstands für das Lernen bereit oder noch nicht geeignet ist.

Die intelligente Unterstützung während des Bearbeitens von Programmieraufgaben hat ELM-ART von seinem Vorgänger ELM-PE [Weber 1994, 1996] übernommen. Der Analyse zugrunde liegt das Episodic Learner Model (ELM), ein kognitives Modell, das individuelle Programmierepisoden im Sinne eines Fallbasierten Systems speichert und in ähnlichen Problemsituationen als Hilfestellung abruft. Neben der Fehlerdiagnose unterstützt ELM somit ein beispielbasiertes Programmieren, indem es relevante Beispiele, die der Lernende selbst erstellt hat oder im Lehrmaterial kennen gelernt hat, präsentiert [vgl. Weber 1994, 8].

---

[36] ELM-ART kann in seiner neusten Version unter der Adresse http://apsymac33.uni-trier.de:8080/elm-art/login-d kostenlos getestet werden. Es ist lediglich eine Registrierung nötig, da ein individuelles Lernermodell für die Adaption des Systems angelegt wird.

Benötigt der Programmiereranfänger während der Aufgabenbearbeitung Hilfe oder ist er sich über seinen Lösungsversuch unsicher, kann er ein Beispiel oder eine Diagnose zu seinem Code anfordern. In abgestuften Schritten werden im Falle eines gefundenen Fehlers – in ELM-PE treten syntaktische Fehler nicht auf, da die Kodierung über einen strukturierten Editor stattfindet[37] – vage Hinweise bis hin zum korrekten Code angeboten. Die ELM-Systeme überlassen damit im Gegensatz zum LISP-Tutor die Kontrolle dem Lernenden und zwingen ihn nicht, einen optimalen Lösungsweg einzuhalten [vgl. Weber 1994, 177]. Die grundsätzliche Funktionsweise der Fehlerdiagnose und das dafür erforderliche Bereichswissen weisen jedoch starke Ähnlichkeit zum LISP-Tutor auf, weniger auch zu PROUST.

LISP-Konzepte und Regeln, welche die Verwendung der Konzepte beschreiben, sind hierarchisch als Frames im Expertenmodell repräsentiert. Konzepte umfassen Wissen über konkrete LISP-Prozeduren oder semantische Konzepte, aber auch „Schemata über allgemeines Algorithmen- und Problemlösewissen“ [Weber 1994, 186]. Ihre Frames enthalten Verknüpfungen zu unter- und übergeordneten Konzepten und semantische Relationen ('is-a'/ 'part-of') sowie mögliche Transformationen in gleichwertige Variationen. Weitere 'Slots' geben das Programmierziel eines Konzeptes und die Regeln zu möglichen Implementierungen an. Regeln spiegeln korrekte, suboptimale oder fehlerhafte Anwendungen wider, womit eine Fehlerbibliothek im Bereichswissen integriert ist [vgl. Weber 1994, 188].

Gemäß dem Model-tracing-Ansatz wird zur Erklärung des aktuellen Codes eines Lernenden versucht, durch Regeln und Plantransformationen diesen Code zu generieren. Zur Erzeugung eines Ableitungsbaums werden auf Basis einer Aufgabenspezifikation, die eine Referenzlösung und Informationen über zu erfüllende Pläne angibt, rekursiv entsprechende Konzepte und ihre Regeln angesprochen[38]. Um den Suchraum einzugrenzen und abwegige Erklärungen durch Kombination vieler seltener fehlerhafter Regeln zu vermeiden, kommen Regeln sortiert nach ihrer Qualität von gut bis fehlerhaft sowie einer zugewiesenen Priorität zur Anwendung. Fehler werden durch den Fehlerregeln angefügte Templates abgestuft rückgemeldet und ggf. Lösungen für ein aktuelles

---

[37] Funktionsaufrufe können aus einem Menü ausgewählt oder über Tastatur eingegeben werden, woraufhin automatisch Schablonen für die nachfolgenden Argumente inklusive Klammern generiert werden. Syntaktisch fehlerhafte Tastatureingaben werden sofort zurückgewiesen [Weber 1994, 7f].

[38] Zuerst findet eine Voranalyse basierend auf TALUS statt, um den grundsätzlich verwendeten Algorithmus und die Reihenfolge der Funktionsaufrufe zu bestimmen. Diese Analyse ist jedoch nicht ausreichend für eine kognitive Diagnose über Art des Fehlers und seine Entstehung [Weber 1994, 192f].

Teilziel benannt. Dabei kennt ELM nicht nur eine ideale Lösung, sondern schlägt – falls möglich – Korrekturen gemäß dem Kontext, innerhalb des tatsächlich begangenen Pfades, vor [vgl. Weber 1994, 178f]. Die so ermittelten angewendeten Konzepte und Regeln mit dem konkreten Code werden daraufhin im episodischen Lernermodell gespeichert, indem sie an den jeweiligen Stellen der Framehierarchie des Domainwissens eingefügt werden[39] (siehe Abbildung 6) [vgl. Brusilovsky & Weber 2001, 367]. Im Lehrmaterial studierte Beispiele werden ebenso den entsprechenden Frames im Lernermodell als Episoden angehängt.

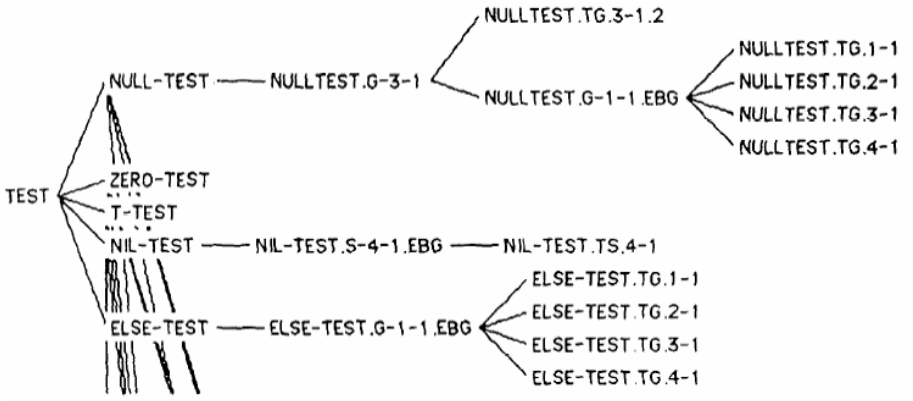

Abbildung 6. Framehierarchie mit Episoden. Aus: Weber 1996, 212

Während der Analyse werden diese individuellen Instanzen vor den anderen Regeln und Konzepten der Wissensbasis gestestet, um den Suchraum einzugrenzen. Eine relevante Verkürzung der Suche ergibt sich erst aus Generalisierungen der Fälle, bei der Schablonen mit Variablen (siehe Abb.7) erzeugt werden, so dass gleiche Regeln und Konzepte ansprechende Code-Muster gegen sie abgeglichen und damit die Ableitungsstruktur direkt übernommen werden kann.

Neben dieser erklärungsbasierten Generalisierung nach der EBG-Methode [Mitchell et al. 1986; siehe Weber 1994, 200f][40] für die ersten Instanzen unter einem Konzept wird in ELM auch ähnlichkeitsbasiert generalisiert, um Gemeinsamkeiten mehrerer Instanzen zu einem Konzept zu erfassen. „Somit gehen in die Generalisierung neben semantischen

---

[39] Brusilovsky & Weber [2001, 367] sprechen hier von 'snippets', da nicht der gesamte Code der Lösung des Lernenden als Fall behandelt wird, sondern jeweils die Codefragmente zu einem Konzept oder einer Regel.

[40] Beim Erklärungsbasierten Lernen wird für ein konkretes Beispiel eine Erklärung generiert, warum dieses Beispiel zu einer Klasse (hier ein Konzept) gehört. Dafür ist eine Wissensbasis über die Domäne nötig, welche durch Integration des neu gelernten Beispiels für eine Klasse erweitert wird. [vgl. Herrmann 1997, 85]. Zur EBG-Methode und dem erklärungsbasierten Abruf von Fällen in ELM siehe Weber [1994, 1996].

Ähnlichkeiten (gemeinsame übergeordnete Konzepte und Regeln) auch strukturelle Übereinstimmungen (über gemeinsame Pläne und Code-Fragmente) ein.“ [Weber 1994, 202].

Some Slots of the Frame NIL-TEST.TS.4-1 (Before Generalization)

Name:	NIL-TEST.TS.4-1
Type:	EPI-Instance
Episode-No:	4
Event-No:	1
Task	Simple-And
Context:	(2.-Case (NIL-TEST ...))
Concept:	NIL-TEST
Abstraction:	NIL-TEST
Transformation:	nil
Plan:	(NIL-TEST (FIRST-ELEMENT (PARAMETER ?LIST)))
Datum:	(equal (car li) nil)
Rule:	Equal-NIL-Test-Rule
Rule-Quality:	suboptimal
Overall-Quality:	suboptimal
Overall-Priority:	6

Some Slots of the Generalization Frame NIL-TEST.S.4-1.EBG

Name:	NIL-TEST.S.4-1.EBG
Type:	EPI-Generalization
Context:	(2.-Case (NIL-TEST ...))
Concept:	NIL-TEST
Abstraction:	NIL-TEST
Datum:	(<Equal-Op> (<First-Elem-Op> <Variable>) nil)
Rule:	Equal-NIL-Test-Rule
Rule-Quality:	suboptimal
Specializations:	(NIL-TEST.TS.4-1)

Abbildung 7. Frame einer Episode vor und nach Generalisierung. Aus: Weber 1996, 210f

Die beschriebenen fallbasierten Techniken zeichnen ELM gegenüber den anderen Systemen aus, da sie zur Reduzierung des Rechenaufwands die Vorhersage eines individuellen Problemlösestils und damit das Abrufen von selbst entwickelten Programmfragmenten sowie dem jeweiligen Stil angemessenen Beispielen erlauben. Dazu wird eine korrekte Lösung für die Problemstellung generiert, als Episode in das Lernermodell eingefügt und die ähnlichsten Fälle abgerufen [Brusilovsky et al. 1995, 328]. Dennoch bleibt der hohe Aufwand für die Modellierung einer umfangreichen – aber stets unvollständigen – Wissensbasis, auf deren Grundlage erst Fälle abgebildet werden [Deek & McHugh 1998, 167].

Systematische und repräsentative Evaluationen zur Zuverlässigkeit der Diagnose sowie zur Akzeptanz von Beispielen als Hilfe werden nicht berichtet, eine Gegenüberstellung zwischen ELM-PE und ELM-ART erwies lediglich einen stärkeren Lernerfolg bei ELM-ART [vgl. auch Deek & McHugh 1998, 158]. Brusilovsky & Weber [2001, 378] erwähnen jedoch, dass Userfeedback zu ELM-ART eine seltene Nutzung von Beispielen während des Problemlösens ergab, sie bei Inanspruchnahme dann aber als sehr hilfreich eingestuft wurden.

Weber [1996, 217-225] weist in einer Studie, in der Anfängerprogramme zur Bildung der Lernermodelle sowie als Testdaten herangezogen werden, nach, dass über episodische Informationen die Anzahl zu testender Regeln und folglich die Diagnosezeit stark verkürzt werden kann. Weber zeigt die Nützlichkeit des Ansatzes zur Vorhersage persönlicher Programmierstile – und damit den Einfluss früherer Beispiele auf neue Situationen [Weber 1996, 225ff]. ELM konnte anhand der erstellten Lernermodelle 51 %

der 238 Programme von 14 Studenten korrekt voraussagen, 91,7 % sogar, wenn diese Lösungen keine Fehler beinhalteten. Weber erklärt die mangelnde Fähigkeit, Fehler zu prognostizieren, dadurch, dass der gleiche Fehler zumeist nicht systematisch wiederholt wird, sondern er sporadisch auftritt. Es ist ihm aber zuzustimmen, dass dies für ein beispielsbasiertes Hilfesystem nicht zwangsläufig als ein Manko anzusehen ist, da eine zur Situation passende korrekte Lösung durchaus ein gutes Analogon darstellen sollte.

3.4.2 Bridge

Im Gegensatz zu den ELM-Systemen legt Bridge [Bonar & Cunningham 1988] eine besondere Betonung auf die Unterstützung des Planungsaspekts der Programmierung, bietet aber zudem Werkzeuge zur Visualisierung der Programmausführung und des Datenflusses. Mit dem Ziel, die Lücke zwischen syntaktischen Ansätzen des Lernens einer Programmiersprache und den nötigen kognitiven Prozessen des algorithmischen Problemlösens zu überbrücken [vgl. Deek & McHugh 1998, 153], kann Bridge den Lernenden in drei getrennten Phasen – der informellen Problemspezifikation, Überführung in Programmpläne und deren Implementierung – coachen.

Über ein Menü werden zuerst zu einer gegebenen Problemstellung zu erreichende Teilziele in einfachem Englisch wie '*Setup the memory variables for the integer and its result*' [nach Ramadhan & Shahib 2000, 166] ausgewählt. Über ein zweites Menü müssen nach erfolgreichem Festlegen der (natürlichsprachlichen) Schritt-für-Schritt-Prozeduren die zugehörigen Pläne als semiformale Ausdrücke z. B. 'Declare Num1' [nach Ramadhan & Shahib 2000, 167] vom Lernenden gefunden werden. Schließlich gilt es, erneut mithilfe eines Menüs die konkreten PASCAL-Code-Fragmente zu diesen Plänen zu bestimmen.

Der Tutor verhält sich in allen Phasen solange passiv bis eine Abweichung von den Vorgaben zu Zielen, Plänen oder Implementierungen in gespeicherten Referenzlösungen auftritt und greift dann durch Hinweise helfend ein. Da Syntaxfehler durch den strukturierten Editor von vornherein abgefangen werden, sind solche Fehler fehlende oder falsch angeordnete Ziele und Pläne sowie falsche Zuordnungen von Zielen zu ihren Plänen bzw. von Plänen zu ihren Sprachkonstrukten. Die Trennung der einzelnen Phasen ermöglicht zum einen, dem Lernenden sich auf den jeweiligen Aspekt des Programmierens zu konzentrieren. Darüber hinaus vereinfacht es die Analyse erheblich. Denn aufwendige Verfahren zum Schließen auf Ziele und Pläne anhand von Code können durch das explizite 'Nennen' umgangen werden [vgl. Ramadhan & Shahib 2000,

167]. Neben dem Tutor kann der Lernende in der Phase der Implementierung die Werkzeuge des Systems nutzen, um schrittweise die Ausführung des Programms und den Datenfluss nach zu verfolgen, wobei die Beziehungen zwischen Plänen und Anweisungen optisch hervorgehoben werden.

Somit fördert Bridge die bewusste Zerlegung von Problemen in Teilaufgaben und das Zuordnen dieser informellen Ziele zu Plänen und deren Code-Mustern, beides wichtige Schritte im Lernprozess [siehe Kap. 2; vgl. Weber 1994, 263; Deek & McHugh 1998, 154]. Dem stehen jedoch einige Nachteile gegenüber: Durch die Vorgabe von Menü-Optionen werden keine komplexeren Strukturen wie Funktionen unterstützt und damit können nur einfache Problemstellungen bearbeitet werden. Vor allem aber berichten Bonar & Cunningham [in Ramadhan & Shahib 2000, 167f] von großen Schwierigkeiten der Lernenden in der Phase der Übertragung von informellen Plänen auf konkrete Sprachkonstrukte. Da Bridge auf das explizite Lehren von Programmierschemata z. B. anhand von Beispielen verzichtet [vgl. Weber 1996, 263], setzt nicht nur die Transformation von Zielen in semi-algorithmische Pläne, sondern gerade dieses Mapping auf Code-Muster ein gewisses Maß an Wissen voraus. Aufgrund der strikten Trennung der Phasen müssen Pläne ausgewählt werden, ohne die Code-Entsprechungen vorher absehen zu können.

So empfehlen Ramadhan & Shahib [2000, 167f] eine stärker aktiv hinführende Rolle des Tutors oder alternativ eine explorative Umgebung, um das Identifizieren von Plänen und ihren Anwendungsmöglichkeiten zu ermöglichen. Einen aktuellen Ansatz für die Umsetzung eines solchen aktiven Tutors, der in einem natürlichsprachlichen Dialog durch Fragen, Hinweise und ggf. Beispielprogramme der Formulierung von Zielen und Plänen dient, die daraufhin in Pseudo-Code umgesetzt werden, stellen Lane & VanLehn [2004] vor. Die Dialogtechniken und Strategien des Tutors beruhen auf einer Korpusanalyse von Tutorien zwischen Anfängern und menschlichen Tutoren.

3.4.3 Discover

Mit der Absicht, die Direktivität bestimmter ITS wie dem LISP-Tutor zu umgehen, ohne dabei Interaktivität einzubüßen, konzipierte Ramadhan [1992; Ramadhan & Shihab 2000] DISCOVER als eine Kombination von freier Programmierumgebung und ITS. Eine 'exploratory phase' ohne Tutor erlaubt dem Programmieranfänger das Experimentieren mit Sprachkonstrukten und vorgefertigten Beispielen, wobei er durch Werkzeuge zum Testen und Visualisieren unterstützt wird. Neben einem strukturierten Editor

bietet das Interface daher Bereiche, in denen die versteckten Prozesse der 'notional machine' sichtbar werden: eine Anzeige aller eingegebenen Werte, die aktuellen Werte von Variablen, die schrittweise Ausführung eines Algorithmus und Ausgaben.

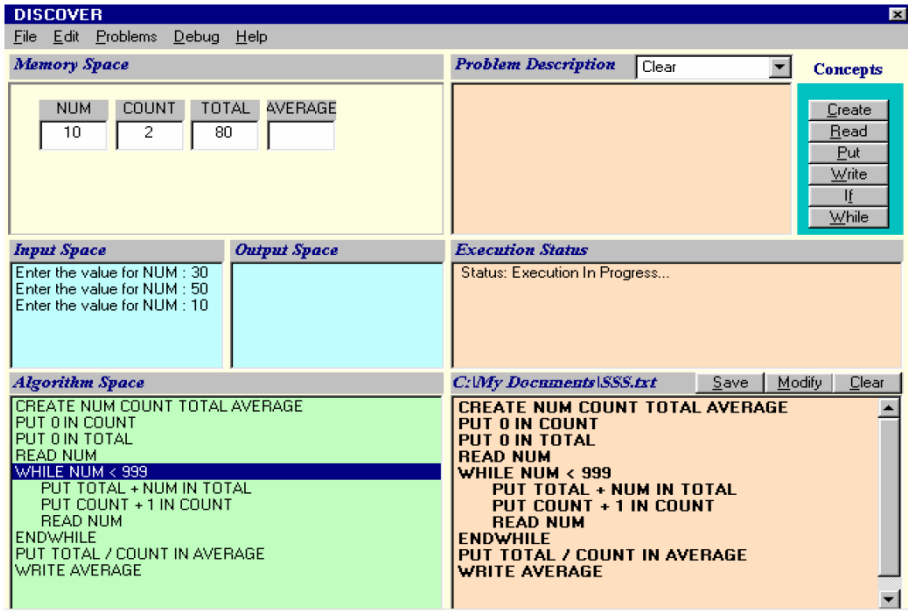

Abbildung 8. Interface des Systems DISCOVER. Aus: Ramadhan & Shihab 2000, 170

Der Editor entlastet von reinen Syntaxschwierigkeiten, indem über ein Menü Konzepte ausgewählt werden, die dann zusammen mit den nötigen Platzhaltern bspw. für Parameter im Code-Bereich angezeigt werden. Allerdings umfasst der realisierte Sprachumfang in DISCOVER bisher keine komplexen Datentypen oder Methoden. In der 'guided phase' wird der Programmieranfänger vom Tutor Schritt für Schritt zur Lösung einer Problemstellung – die richtige Zusammenstellung von Konzepten und ihre Erweiterung - geführt, wobei ihm weiterhin die Visualisierungswerkzeuge der Plattform zur Verfügung stehen. Feedback erfolgt durch Erklärungen im Falle eines Fehlers, aber auch bei erfolgreichem Meistern eines Schrittes als Erinnerung und Hinweis bezüglich ausstehender Ziele. Entsprechend des Model-tracing-Ansatzes wird der Programmierer dabei auf einem richtigen Pfad zur Lösung gehalten, jedoch ermöglicht eine geringere Granularität – Analyseeinheiten sind ganze Zeilen/ Anweisungen und nicht einzelne Symbole wie im LISP-Tutor – eine immerhin eingeschränkt selbstständige Fehlerkorrektur. Im Gegensatz zu anderen intelligenten System verfügt DISCOVER derzeit über kein Expertenwissen, kann daher Problemstellungen nicht generativ über Regeln lösen und nicht erklären, wieso eine bestimmter Schritt an einer Stelle nötig ist [vgl. Ramadhan & Shihab 2000, 162].

Als Wissensbasis zur Fehlererkennung nutzt das System eine Referenzlösung, zu der von Hand Ziele (entsprechen den Konzepten des Menüs), Pläne (Code-Fragmente) sowie Erklärungen für die Kommunikation des Tutors kodiert werden. In Anlehnung an PROUST lässt sich das Musterprogramm so als ein 'goal-and-plan-tree' mit AND/ OR-Verknüpfungen darstellen, bei dem Ziele z. B. die Berechnung eines Durchschnitts auf verschiedene Arten in Plänen implementiert werden können. Hier sind jedoch die Beziehungen zwischen den Zielen und Unterzielen als Bedingungen modelliert, wodurch Ziele gemäß dem Model-tracing in vorgegebener Reihenfolge des Pfades kodiert werden müssen. Die Menüauswahl von Konzepten fördert dabei nicht nur explizites Planen auf Seite der Lernenden, sondern sie teilen zudem dem System die aktuellen Ziele mit, so dass eine aufwendige Interpretation z. B. mithilfe von Planimplementierungen und Transformationen (PROUST) in der Diagnose umgangen werden kann. Fehler in der Reihenfolge nötiger Teilschritte zur Lösung eines Problems erfordern somit keine weitere Analyse.

Fehler in der Implementierung eines Ziels werden durch den Abgleich zu den entsprechenden Code-Fragmenten der Referenzlösung über Pattern-Matching und Heuristiken ermittelt. Da DISCOVER als Analyseeinheit eine ganze Anweisung und nicht einzelne Symbole heranzieht, kann in bestimmten Fällen – aufgrund einer geringeren Anzahl von zu antizipierenden Pfaden - dabei der Rechenaufwand verglichen zu Produktionssystemen mit Regeln erheblich reduziert werden. Die größeren Analyseeinheiten ermöglichen neben der Chance zur Selbstkorrektur ein sinnvolleres Feedback für den Lernenden, da einige Fehlermeldungen erst im Kontext der gesamten Anweisung verständlich sind [vgl. Ramadhan & Shihab 2000, 180]. Eine Evaluierung an acht Studenten zeigte, dass Lernende mit dem System Problemstellungen schneller und sorgfältiger lösten als jene in der Kontrollgruppe ohne intelligentes System [vgl. Deek & McHugh 1998, 157].

Zusammenfassend lässt sich sagen, dass für die Unterstützung des Programmierprozesses bei Anfängern verschiedene Techniken zum Einsatz kommen. Diese lassen sich sowohl didaktisch nach dem Grad der Hilfestellung als auch technisch nach dem für die Analyse eingesetzten Wissen unterscheiden. Während Programmierwerkzeuge wie Visualisierung, Textstrukturierung oder Beispiele Fehler nicht explizit aufdecken, sondern zur Erleichterung einer selbstständigen Korrektur durch die Lernenden dienen,

zeigen automatische Debugger Fehler direkt auf. Syntaktische Fehler können durch das Prüfen auf typische Anfängerfehler, die Klassifizierung von Compiler-Meldungen oder Parser-Repair-Verfahren identifiziert werden. Semantische oder logische Fehler lassen sich durch automatisierte Testfälle oder die Berechnung der Gleichwertigkeit zwischen einem Lösungsversuch und einer Referenzlösung erfassen.

Da diese Techniken jeweils nur bestimmte Fehlertypen erkennen oder zulässige Variationen als inkorrekt einstufen, modellieren ITS umfangreiches Programmierwissen zumeist mit zusätzlichen Fehlerkatalogen, um Fehler exakter identifizieren und ihre Ursachen erklärend benennen zu können. Aufgrund der Evaluierungsergebnisse ist jedoch fraglich, ob der intellektuelle Aufwand für die Erstellung einer Wissensbasis, deren Vollständigkeit nicht gewährleistet werden kann, im Verhältnis zu der Wirksamkeit der Hilfestellung steht. Laut Deek & McHugh [1998, 160-174] reicht die alleinige Unterstützung der Implementierungsphase nicht aus, da Anfängerprobleme darüber hinaus auch Planung und Design des Programms betreffen.

Um bewerten zu können, welche Unterstützungsformen für den Einsatz in VitaminL sinnvoll sind, widmet sich die nun folgende Analyse der Identifizierung möglicher Anfängerprobleme bei der Programmierung in virtuellen Teams.

4 EMPIRISCHE STUDIE: JAVA-ANFÄNGERFEHLER

Die zahlreichen Untersuchungen von Programmieranfängern haben verschiedene typische Anfängerfehler und -probleme aufgezeigt. Jedoch sind Anfänger dabei zumeist in ihrem individuellen Problemlöseprozess betrachtet worden. In der folgenden empirischen Untersuchung soll überprüft werden, ob sich die in der Forschung bisher ermittelten Probleme auch auf die spezielle Situation des Lernens von Java in virtuellen Teams übertragen lassen bzw. welche Probleme darüber hinaus entstehen. Dazu folgt in den nächsten Abschnitten eine Beschreibung der Daten und der Auswertungsmethodik, um daraufhin die ermittelten Fehler und Probleme zu diskutieren.

4.1 BESCHREIBUNG DER DATEN

Um typische Problemsituationen bei der Java-Programmierung in virtuellen Teams modellieren zu können, wurden Studenten beim virtuellen Bearbeiten von Übungen über die VitaminL-Plattform in Form von Logfiles aufgezeichnet. Mithilfe dieser Logfiles im XML-Format ist es möglich, komplette Programmiersitzungen mit ihrer Kommunikation, Kodierung und Interaktion nachträglich in Echtzeit oder auch im Zeitraffer nachzuverfolgen [vgl. Kölle & Langemeier 2004a]. Analysiert wurden auf der Basis dieser Protokolle Hildesheimer und Konstanzer Studenten, die in gemischten Teams während eines virtuellen Kooperationskurses zur Einführung in die Programmiersprache Java im Sommersemester 2005 online Aufgaben lösten. Da die so erworbenen Felddaten nur eingeschränkt valide und repräsentativ sind, wurden zusätzlich Benutzertests mit Hildesheimer Studenten durchgeführt, die ebenfalls einen Java-Einführungskurs belegten, jedoch traditionell als Vorlesung.

4.1.1 Felddaten aus einer virtuellen Kooperationsveranstaltung

Die hier analysierten Teams setzen sich aus Studenten der Universität Hildesheim und der Universität Konstanz zusammen, die im Sommersemester 2005 an der virtuellen Kooperationsveranstaltung beider Universitäten zur Einführung in die Programmiersprache Java teilnahmen. In der Präsenzphase wurde dabei die Vorlesung per Videokonferenz nach Konstanz übertragen und deren Inhalte in einer Online-Übung in Dreier- oder Viererteams praktisch angewandt.

Die 33 Teilnehmer bildeten fünf gemischte Teams aus Hildesheimer und Konstanzer Studenten und zwei Teams nur aus Konstanzer Studenten. Die Hildesheimer Teammitglieder als Studenten des Internationalen Informationsmanagements im zweiten Semester waren Programmieranfänger und verfügten im Durchschnitt über nicht mehr Grundlagen als über die aus der Einführung in die EDV. Dagegen waren die Konstanzer Teilnehmer keine Anfänger. Bei den meisten handelte es sich um Studierende des Master-Studienganges Information Engineering[41] mit stark informatikorientiertem Profil und soliden Vorkenntnissen in C++.

Die zu bearbeitenden Aufgaben[42] waren so konzipiert, dass sie die Anwendung grundlegender Programmier- und Sprachkonzepte wie Variablen, Bedingung bzw. Kontrollfluss, Iteration, Methoden, Objekte und gängige Code-Muster im Sinne der bereits beschriebenen Pläne erforderten. Dazu sollten anfänglich kleinere Programme zur Berechnung des Wechselgeldes eines Getränkeautomaten, ein Pasch-Würfelspiel gegen den Computer, die Mittelwertbildung eingegebener Zahlen[43], die Ermittlung der Tageszahl eines als Parameter übergebenen Monats oder eine Textenkodierung bzw. -dekodierung entwickelt werden.

Später im Kurs umfassten die Aufgaben mit der Behandlung der Objektorientierung und ihrer Prinzipien z. B. die Modellierung eines einfachen Autohauses oder einer Bibliothek. Für das Autohaus waren einzelne Klassen für verschiedene Fahrzeuge wie 'Auto', 'Lkw' oder 'Motorrad' mit ihren typischen Merkmalen und eine Methode zur Ausgabe dieser Objekte mit ihren Werten zu programmieren. Ähnlich waren für eine Bibliothek, nach Vorgabe einer genauen Spezifikation anhand eines UML-Klassendiagramms, die Klassen 'CD' und 'Buch' zu implementieren. Die Klassen waren darauf aufbauend in weiteren Aufgaben bezüglich einer Oberklasse mit gemeinsamen Attributen, einer Erweiterung zum Einlesen konkreter Medienobjekte aus einer Datei und dem Schreiben von Medien in eine Datei abzuändern. In einer weiteren Aufgabe zur GUI-Programmierung sollte ein erstes einfaches Fenster erzeugt werden. Die letzten im Rahmen der Analyse betrachteten Aufgaben erforderten komplexere Programme zur Erstellung einer Textstatistik mit Worthäufigkeiten und die Erkennung von Eigennamen aus eingelesenen Dateien.

---

[41] Der Aufbau des Studiums ist ausführlich beschrieben unter http://www.uni-konstanz.de/studium/pdf/stuplaene/J126_ST_infengin.pdf

[42] Eine genaue Auflistung aller Aufgaben ist der beigelegten CD zu entnehmen.

[43] Eine typische Aufgabe für Programmieranfänger, die der viel zitierten 'Rainfall'-Aufgabe sehr ähnelt. Diese diente in vielen Studien der Untersuchung von Anfängerproblemen.

Die Repräsentativität und Validität der so erhobenen Felddaten für die Modellierung von Problemsituationen von Programmieranfängern in virtuellen Teams wird jedoch durch mehrere Faktoren beeinträchtigt. Die Übungen waren insofern eingeschränkt virtuell, als dass sie im Anschluss an die Vorlesung in tatsächlicher Anwesenheit eines menschlichen Tutors in Hildesheim gehalten wurden. Die jeweils in Hildesheim oder Konstanz sitzenden Teilnehmer konnten sich per Kamera sehen. Aus den entsprechenden Logfiles lässt sich zwar die gesamte Kodierung und Chat-Kommunikation zwischen Hildesheimer und Konstanzer Teammitgliedern, nicht aber die Kommunikation zwischen den Teammitgliedern in einem Raum rekonstruieren. Dennoch stammt der Großteil der Logfiles von einer rein virtuellen Hausaufgabenbearbeitung der Teams zwischen den Vorlesungen, bei denen sich gelegentlich ein Tutor zugeschaltet hat.

Außerhalb der VitaminL-Plattform wurden von den Teilnehmern andere Tools zur Kommunikation und Programmierung genutzt. Dies kann zurückgeführt werden auf technische Schwierigkeiten zu Beginn des Kurses und vermutlich auch auf eine dadurch empfundene Erleichterung der gemeinsamen Arbeit. So wurde neben dem VitaminL-Chat via Instant-Messenger oder Internet-Telefonie, wie ICQ und Skype, kommuniziert. Das führt wiederum zu einer Reduzierung des Informationsgehalts der analysierten Logfiles, die zwar Hinweise über die Verwendung anderer Tools geben aber keine genauen Rückschlüsse über den konkreten Anteil zusätzlicher Kommunikation zulassen[44].

Ähnlich verhält es sich mit anderen Programmierplattformen und Hilfsmitteln zur Überwindung von potentiellen Problemsituationen[45]. Aufgaben wurden häufig 'extern' – vermutlich mit mächtigeren Entwicklungswerkzeugen wie Eclipse – vorbereitet und zur Sitzung zusammen mit dem Team besprochen und ggf. debuggt. Neben den erwähnten technischen Schwierigkeiten ist als weitere Ursache dafür das Leistungsgefälle innerhalb der gemischten Teams anzunehmen. Denn die Konstanzer Teilnehmer mit ihren fortgeschrittenen Programmierkenntnissen übernahmen während der Arbeit in VitaminL einen Großteil der Implementierung – mit Auswirkungen auf die ermittelten Fehler und ihre Häufigkeiten.

---

[44] Beispiele für solche Hinweise:
2005.05.07-10.55.15.log.xm: [T] habt ihr eigentlich skype??; … [W] Aim oder Icq? … [r] mom muss skype noch laden
2005.05.23-14.02.41.log.xml: [S] seid ihr im Skype?
2005.05.04-17.01.06.log.xml: [C] aber die tina erklärt grad dem roland alles über skype

[45] 2005.06.08-15.00.49.log.xml: [j] sind noch fehler drin, hab gra din eclipse noch mal geguckt, aber nur rechtschreibung.monent

In die Analyse eingegangen sind 23 in ihrem Umfang stark variierende Logfiles aus den Wochen zwei bis neun des Kurses, in denen 14 verschiedene Aufgabenstellungen bearbeitet wurden. Zur Erfassung möglichst vielfältiger Fehler und Problemsituationen wurde die Auswahl unterschiedlicher Aufgaben bevorzugt, so dass nur wenige Aufgabenstellungen zwischen mehreren Gruppen vergleichend analysiert wurden. In den seltensten Fällen beinhaltet ein Logfile den Gesamtprozess von Planung, Spezifikation, Implementierung und Debugging, da Aufgaben vorbereitet oder Sitzungen abgebrochen worden sind. Aufgrund der unzureichenden Datenmenge aus rein virtuellen Settings wurde entschieden, auch Logfiles heranzuziehen, die während der Übung aufgezeichnet wurden, daher aber weniger Kommunikation enthalten.

Folglich bilden die vorliegenden Daten lediglich einen Ausschnitt des gesamten Problemlöseprozesses ab. Es ist davon auszugehen, dass in reinen Anfängerteams weitaus mehr Fehler und Problemsituationen ermittelt worden wären. Aus diesem Grund wurden zusätzlich Daten mit Hildesheimer Studenten erhoben, die im gleichen Zeitraum in Hildesheim die Einführung in Java als traditionelle Vorlesung mit Begleitübung besuchten. Trotz der benannten Einschränkungen sollten diese so erhaltenen Logfiles eine gute Grundlage für die explorative Untersuchung typischer Problemsituationen bilden.

4.1.2 Benutzertests

In den Benutzertests bearbeiteten freiwillige Teams zu je drei Studenten des zweiten Semesters Internationales Informationsmanagement jeweils über einen Zeitraum von etwa 90 Minuten eine ihnen unbekannte Aufgabe. Bei den Studenten handelte es sich fast ausschließlich um Programmieranfänger, wie ein Fragebogen zur Selbsteinschätzung ihrer Java-Kenntnisse und zu Kenntnissen weiterer Programmiersprachen zeigte. Von den 15 Teilnehmern der fünf ausgewerteten Benutzertests verfügte ein Teilnehmer über 0,5 Jahre Erfahrung in Prolog und ein weiterer über vier Jahre HTML sowie ein Jahr Turbo-Pascal. Nur ein Teilnehmer bewertete seine Java-Kenntnisse auf einer fünfstufigen Ordinalskala als 'Sicher in Inhalten der Vorlesung' (3), während neun Personen 'Einige Unsicherheiten' (4) angaben und fünf sie als 'gering' (5) einstuften.

Die Inhalte der traditionellen Vorlesung deckten sich im Wesentlichen mit denen der virtuellen Veranstaltung und die in den Übungen zu lösenden Aufgaben waren ähnlich konzipiert. In den betreuten Übungen erarbeiteten die Gruppen mithilfe des RealJ-Editors und u. U. der Eingabeaufforderung für Kompilieren und Ausführen ihre Programme zusammen an einem Computer. Als Motivation für die Teilnahme wurde ihnen

als Alternative zur bekannten Arbeitsweise das Bearbeiten einer Aufgabe[46] über die VitaminL-Plattform unter tutorieller Unterstützung und eine anschließende Beratung zu anderweitigen Kursaufgaben angeboten.

Zur Bestimmung der angemessenen Aufgabenschwierigkeit und der Planung des zeitlichen Rahmens wurden mehrere Pretests durchgeführt. In einer ersten Variante sollten die Gruppen für ein Möbelhaus zu einer vorgegebenen Klasse 'Möbel' die Unterklassen 'Bett' und 'Kommode' mit ihren Attributen, Getter- und Settermethoden modellieren und berechnen, welchen Einkaufswert alle Möbel einer bestimmten Marke haben. Da die Objektorientierung mit ihren Konzepten zum Zeitpunkt des Tests gerade erst Inhalt der Vorlesung war und daher noch nicht in einer Übung praktisch angewendet worden waren, hatten die Testpersonen so große Schwierigkeiten, dass in der angesetzten Zeit kaum zu analysierender Code produziert wurde. Beschäftigt damit, in Vorlesungsunterlagen und Lehrbüchern nach Beispielen und Hilfestellungen zu suchen, kommunizierten die Teammitglieder nur wenig miteinander.

Ein weiterer Pretest, in dem einfache Personenobjekte mit Namen nach Muster eines angegebenen Beispiels für einen alphabethischen Stringvergleich sortiert werden sollten, bestätigte die Unangemessenheit einer objektorientierten Aufgabenstellung zum momentanen Kenntnisstand nach sieben bis acht Wochen Kursteilnahme. Daher wurde die vorgesehene Untersuchung speziell objektorientierter Konzepte zugunsten einer Aufgabenstellung verworfen, deren zur Lösung nötigen Konstrukte und Pläne großteils schon einmal in den Übungen verwendet worden waren. Es sollten Namen mithilfe einer gegebenen und bereits bekannten Klasse 'Tastatur' von der Konsole eingelesen und je nach Benutzereingabe alphabethisch auf- oder absteigend sortiert und ausgegeben werden. Um diese nicht triviale Aufgabe etwas zu vereinfachen und der kurzen Zeit anzupassen, diente als Hinweis ein Verweis auf die Methode 'compareTo()' der Klasse 'String', mit der zwei Wörter alphabethisch verglichen werden[47] können.

Bevor diese Aufgabe in den Tests bearbeitet wurde, erfolgte eine kurze Erläuterung des Ablaufs durch den Versuchsleiter, das Ausfüllen des Fragebogens zur Selbsteinschätzung sowie eine Einweisung in die Benutzung der VitaminL-Plattform, die den Teams bis dahin unbekannt war. Die Testpersonen wurden unterrichtet, dass sie die gestellte Aufgabe als Team lösen sollten, wobei sie Hilfsmittel wie Lehrbücher, Vor-

[46] Mit der Möglichkeit der Anrechung der Leistung auf das Punktekonto des Kurses.

[47] Der exakte Text der Aufgabe ist auf der beiliegenden CD einzusehen.

lesungsunterlagen oder Internetquellen beliebig nutzen könnten. Sie könnten den virtuell anwesenden Experimentleiter und Tutor bei Problemen jederzeit um Hilfe bitten, indem sie eine Nachricht direkt an ihn adressieren.

Das Untersuchungsziel der Erfassung typischer Problemsituationen wurde den Teilnehmern dabei explizit genannt, um darauf basierend automatisch Hilfestellung bspw. in Form eines elektronischen Tutors bieten zu können. So wurde auch die Bitte um Kompilieren und Ausführen ausschließlich innerhalb der VitaminL-Plattform sowie das Nachfragen beim Tutor begründet, da es sich dabei um Messgrößen für die Auswertung handelt. Nach Instruktion und Einführung in die Plattform wurden die einzelnen Teammitglieder auf verschiedene Räume verteilt, da sich in den Pretests zeigte, dass die virtuelle Zusammenarbeit bei tatsächlicher Anwesenheit der anderen in einem Raum künstlich wirkt und von den Teilnehmern als belastend empfunden wird.

Die Daten aus den letztlich fünf ausgewerteten Benutzertests besitzen somit gegenüber den realistischen Felddaten einige Vorteile: Es waren nur Programmieranfänger beteiligt und externe Werkzeuge zur Kommunikation oder Programmierung wurden mit großer Sicherheit nicht genutzt. Zudem konnte der Entwicklungsprozess von Anfang an – inklusive der Planungsphase – abgebildet werden, wenn es auch keinem Team in der kurzen Zeit gelang, die Aufgaben fertig zu stellen.

Schwierigkeiten mit der Verwendung einer neuen Plattform und einer ungewohnten Arbeitsform erforderten jedoch viele technische Nachfragen, erhöhte Konzentration bei der Benutzung der Plattform und kosteten Zeit. Auch ist eine Beeinflussung durch die künstliche 'Laborsituation' nicht auszuschließen, da der Tutor bei offensichtlichen Stocksituationen im Hinblick auf die beschränkte Zeit auch von sich aus mit strategischen Fragen oder Hinweisen eingriff. Der Frage, welche Merkmale in diesem Zusammenhang eine solche Problemsituation konstituieren können, wird sich die folgende Darstellung der Auswertungsmethodik annähern.

4.2 METHODIK DER ANALYSE

Während viele Studien durch Umfragen, Auswertung vollständiger Lösungsversuche oder kontrollierte Tests zu einer konkreten Fragestellung bestimmte Ausschnitte untersuchen, liefern die vorliegenden Daten trotz der geschilderten Einschränkungen interessante Einblicke in den Gesamtprozess der Programmentwicklung von Anfängern. So kann nicht nur beobachtet werden, welche Fehler häufig auftreten, sondern auch wie

vom Team im weiteren Verlauf mit ihnen umgegangen wird und ob sie zu einer Problemsituation führen. Um die Erfassung dieser Prozesshaftigkeit der Daten für die Analyse von Fehlern und der Modellierung von Problemsituationen zu verdeutlichen, folgt den Erklärungen der zentralen Begriffe die Erläuterung der Vorgehensweise. Danach wird speziell auf die Klassifikation der ermittelten Fehler eingegangen.

4.2.1 Zentrale Begriffe

Fehler wird hier umfassend definiert als Code, der gegen die syntaktischen oder semantischen Konventionen von Java verstößt oder nicht zu einer möglichen Lösung der Aufgabenstellung – wenn auch nicht der effizientesten – führt. Während Fehler, die innerhalb weniger Sekunden behoben werden, nicht als Fehler angesehen werden, gehen Tippfehler im Sinne des Falschschreibens eines Variablen- oder Methodennamens in den Fehlerkatalog ein[48]. In Bezug auf die Erfüllung der Aufgabe wird auch eine fehlende Implementierung von Programmspezifikationen[49] bspw. das fehlende Prüfen auf einen negativen Eingabewert als Fehler gewertet, da sie eine Fehlfunktion des Programms oder Laufzeitfehler wie eine 'ArrayIndexOutOfBoundsException' bewirken können.

Fehler geben einen ersten Eindruck darüber, welche javaspezifischen Konzepte und Notationen sowie welche allgemeinen Programmierkonzepte Anfängern Schwierigkeiten bereiten. Sie sind jedoch noch nicht mit Problemsituationen gleichzusetzen, denn nicht jeder Fehler ist als eine Problemsituation anzusehen und nicht alle Problemsituationen manifestieren sich in direkt beobachtbaren Fehlern im Quelltext.

Eine Problemsituation wird in Anlehnung an den Begriff des Problems [vgl. Zimbardo 1995, 375ff] und an VanLehn's 'impasse' [vgl. VanLehn 1990, 42f] als eine Stocksituation definiert, in der ein Lernender „nicht weiter weiß" [Möbus & Schröder 1994] und somit die Ausführung der aktuellen Handlung nicht reibungslos fortgesetzt werden kann. Innerhalb des Problemlöseprozesses der Programmentwicklung sieht der Lernende sich mit einem Problem konfrontiert, da ihm die Operationen zur Erreichung des gegenwärtigen Ziels bzw. Teilziels nicht verfügbar sind. Das Problem kann aber auch darin bestehen, dass „Information zur Auswahl eines Ziels oder zur Bildung eines

---

[48] Hierbei kann es sich in beiden Fällen um 'Schnitzer' bei der Ausführung einer Routinehandlung [vgl. Grams 1990, 19] oder um den lockeren Umgang mit einer Programmiersprache [Soloway & Spohrer 1986a, 631] handeln. Da hier nicht in erster Linie eine eindeutige Ursachenzuschreibung im Mittelpunkt steht, werden sie als Fehler weiter untersucht, da nicht auszuschließen ist, dass sich aus ihnen Problemsituationen entwickeln (z. B. in Kombination mit geringem Verständnis der Compiler-Meldung '`cannot resolve symbol`').

[49] Dies entspricht der Fehleridentifizierung anhand von Plandifferenzen bei Soloway & Spohrer [1986a]

Plans“ [Möbus & Schröder 1994] fehlt, ein Plan nicht durchführbar ist, keine Bewertungskriterien für einen Plan vorliegen oder das Problemlöseergebnis z. B. in Form eines nicht behebbaren Fehlers unbefriedigend ausfällt [vgl. Möbus & Schröder 1994].

Um den Konflikt zu lösen und fortfahren zu können werden häufig domänenunabhängige und in diesem Sinne wissensarme Heuristiken eingesetzt: jemanden um Hilfe bitten, Inferieren bspw. aus der natürlichen Sprache [siehe Kap. 2.3], 'Versuch-und-Irrtum' oder das Heranziehen oberflächlich ähnlicher Beispiele [vgl. Möbus & Schröder 1994; Edelmann 2000]. Während solche allgemeinen Problemlösestrategien zum Ziel und neuem Wissenserwerb führen können [vgl. Möbus & Schröder 1994], bergen die Problemsituationen gleichzeitig Frustrationspotentiale und die Gefahr, dass bspw. fehlendes Wissen nicht erworben wird [siehe Kap. 2; Ben-Ari 1998].

Diese individuellen Strategien zur Bewältigung einer Stocksituation werden auf den Kontext der Programmierung in virtuellen Teams übertragen. Die Besonderheit gegenüber dem individuellen Problemlöseprozess besteht jedoch darin, dass Stocksituationen eines Einzelnen durch den Transfer von Wissen zwischen den Teammitgliedern unmittelbar überwunden werden können. Daher werden nur Situationen als problematisch betrachtet, in denen die Gruppe als ganzes – bspw. in Folge fehlenden Programmierwissens oder unzureichenden Wissenstransfers – ins Stocken kommt und 'schwache' [vgl. Möbus & Schröder 1994] Techniken anwendet. Da sich das Team nicht selbst helfen kann, ist an dieser Stelle Unterstützung von außen sinnvoll.

4.2.2 Vorgehensweise bei der Auswertung

In der Analyse wurden die 28 Sitzungen mithilfe eines Logfileanalyzers rekonstruiert. Dieser Logfileanalyzer bildet alle Aktionen einer Sitzung auf der VitaminL-Plattform ab, die in den XML-Dateien gespeichert sind, so dass Schritt für Schritt der Ablauf nachgestellt werden kann. Das Öffnen, Speichern, Schließen, Anfordern, Freigeben, Kompilieren und Ausführen von Java-Dokumenten, die Kodierung der einzelnen Teammitglieder als je einzelnes Zeichen und die Kommunikation untereinander werden dabei zusammen mit Informationen über ihre Entstehungszeit angezeigt. Für die Auswertung ist somit ermittelbar, wie viel Zeit nach dem Kompilieren zur Behebung eines Fehlers benötigt wurde[50]. Abbildung 9 stellt die Ansicht dar, die der Logfile-Analyzer auf die Vorgänge während einer Sitzung bietet. Das aktive Logfile-Fenster gibt zudem Informa-

[50] Nicht bei den ersten Dateien, da anfangs nicht in VitaminL möglich.

tionen, die sich kaum in der Plattform darstellen lassen, wie das Navigieren eines Teammitglieds innerhalb oder zwischen Dokumenten.

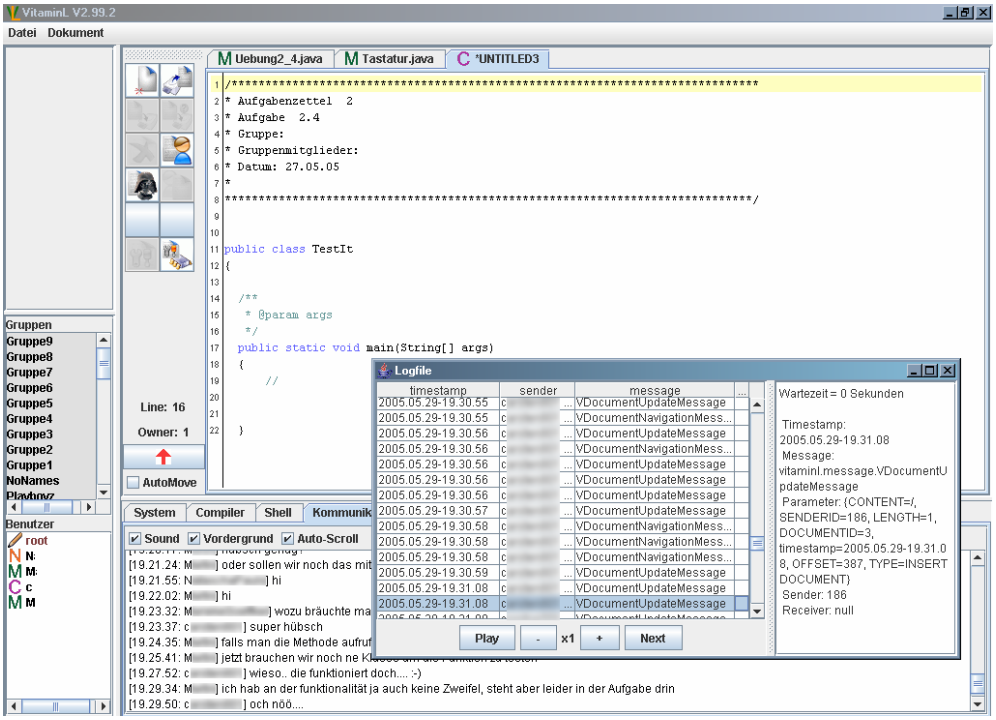

Abbildung 9. Logfile-Analyzer

Beim Nachstellen der Sitzungen wurde jeder auftretende Fehler mit dem entsprechenden Code und Informationen über die Entstehungszeit (timestamp), den Urheber und die aktuelle Aufgabe protokolliert. Falls verfügbar, wurde im weiteren Verlauf zudem notiert, wann, durch wen und durch welchen Code der Fehler behoben wurde und ob bezüglich des Fehlers kommuniziert wurde. Nach Auswertung aller Logfiles erfolgten eine Klassifikation der gefundenen Fehler gemäß dem im nächsten Kapitel beschriebenen Schema und die Ermittlung der Häufigkeiten für Fehlertypen.

Begangene Fehler wurden im nächsten Schritt daraufhin untersucht, ob sie eine Problemsituation darstellten und welche fachlichen Problemsituationen auftraten, die sich nicht einem konkreten Fehler zuordnen lassen. Zur Auswertung der Daten wird der Begriff einer Problemsituation durch mehrere Indikatoren operationalisiert: Zeit, Fragen an den Tutor – sofern anwesend –, die Häufigkeit der Versuche zur Behebung eines Fehlers und Hinweise in der Kommunikation.

Ähnlich vieler empirischer Studien, bspw. Studien der Sozialforschung und Usability-Studien, die Reaktionszeit oder Reaktionsdauer als Indikator für hypothetische Konstrukte messen, dient die Zeit, in der keine Kodierung bzw. keine Kommunikation im Hinblick auf die Programmentwicklung stattfindet, als ein Hinweis auf eine Problem-

situation. Aufgrund der unkontrollierten (Feld-)Daten und der ungewohnten Arbeitsform in den Benutzertests ist Zeit nur ein schwacher Indikator[51]. War ein Tutor bei einer Sitzung anwesend, werden Fragen an den Tutor als weiterer Indikator genutzt, wobei jedoch anzunehmen ist, dass nicht bei jedem Problem direkt nachgefragt wird. Entsprechend der Strategie von 'Versuch-und-Irrtum' werden mehrere erfolglose Anläufe, einen Fehler zu beheben[52] als problematisch gewertet. Zusätzlicher Indikator ist Kommunikation[53], die eine Stocksituation anzeigt bspw. wenn das Team erwägt, einen Tutor zu kontaktieren. Da die Einzelindikatoren für diese Datenbasis teils schwach teils zu selten verfügbar sind, werden alle in einer Situation präsenten Indikatoren für eine qualitative Klassifizierung als Problemsituation kombiniert.

4.2.3 Klassifikationsschema

In den zahlreichen Studien zu Programmierfehlern werden in Abhängigkeit von der zentralen Forschungsfrage, dem Untersuchungsdesign und dem untersuchten Personenkreis entsprechend viele verschiedene Klassifikationsschemata vorgestellt. Einen repräsentativen Ausschnitt oft zitierter Klassifikationen zeigt Abbildung 10.

Diese und weitere Klassifikationsschemata wurden auf ihre Beschreibungsfähigkeit für die extrahierten, fehlerhaften Code-Fragmente geprüft. Die induktive Vorgehensweise zur Klassifikation wurde gewählt, um angesichts der fehlenden Modelle für die Auswertung vergleichbarer Daten in der qualitativen Exploration nicht im Vorfeld eingeschränkt zu werden. Aus den gefundenen Fehlern ergaben sich dabei als Kriterien für ein geeignetes Kategoriensystem: Aufgabenunabhängigkeit, konzeptionelle Klassifikation sowie eine für Anfängerfehler aussagekräftige Granularität.

---

[51] Es kann kein Grenzwert für verstreichende Zeit festgelegt werden, nach dessen Überschreitung auf das Bestehen eines Problems geschlossen werden kann, da eine zwischenzeitliche 'Fremdbeschäftigung' einzelner Teammitglieder während der virtuellen Sitzung auch nicht auszuschließen ist und dies Auswirkungen auf den zeitlichen Aspekt hat.

[52] Der heuristisch festgelegte Wert ist hier drei.

[53] Dabei wird weder eine vollständige Inhaltsanalyse noch eine Diskursanalyse vorgenommen, sondern nur fragmentarisch Anhaltspunkte für fachliche Probleme abgeleitet. Welche speziellen Probleme aufgrund des Kommunikationsverhaltens bei der Programmierung in virtuellen Teams entstehen können, zeigt Göldner [2005].

Study Details	Bug / Error / Cause	Description	Authors' comments
Gould [10], novice, Fortran *1975*	Assignment bug	Errors assigning variables values	Requires understanding of language & behavior
	Iteration bug	Errors iterating	Requires only an understanding of the language
	Array bug	Errors accessing data in arrays	
Eisenberg [7], novice, APL 1983	Visual bug	Clustering semantically related parts of expression	
	Naive bug	Using branching & iteration instead of parallel processing	"…because of need to think step-by-step"
	Logical bug	Omitting or misusing logical connectives or relationals	
	Dummy bug	Experience with other languages interfering	"…seem to be syntax oversights"
	Inventive bug	Inventing syntax	
	Illiteracy bug	Difficulties with order of operations	
	Gestalt bug	Not foreseeing side effects of commands	"…failure to see the whole picture"
Johnson et al. [14], novice, Pascal *1983*	Missing	Omitting required program element	Errors have contexts: input/output, declaration, initialization and update of variables, conditionals, scope delimiters, or combinations of these contexts.
	Spurious	Including unnecessary program element	
	Misplaced	Putting necessary program element in wrong place	
	Malformed	Putting incorrect program element in right place	
Sphorer and Soloway [19]; novice, Basic *1986*	Data-type inconsistency problem	Misunderstanding differences between data types	"All bugs are not created equal. Some occur over and over again in many novice programs, while others are more rare…Most bugs result because novices misunderstand the semantics of some particular programming language construct."
	Natural language problem	Appling natural language semantics to commands	
	Human-interpreter problem	Assuming computer has similar interpretation of code	
	Negation & whole-part problem	Difficulties with constructing logical Boolean statements	
	Duplicate tail-digit problem	Incorrectly typing constant values	
	Knowledge interference problem	Domain knowledge interfering w/ entering constants	
	Coincidental ordering problem	Malformed statements produced correct output	
	Boundary problem	Not anticipating problems with extreme values	
	Plan dependency problem	Unforeseen dependencies in program statements	
	Expectation & interpretation problem	Misunderstandings of the problem specification	
Knuth [15], in writing TeX in SAIL & Pascal *1989*	Algorithm awry	Improperly implemented algorithms	"method proved to be incorrect or inadequate"
	Blunder or botch	Accidentally writing code not to specifications	"not… enough brainpower left to get the…details"
	Data structure debacle	Errors using and changing data structures	"I did not preserve the appropriate invariants"
	Forgotten function	Missing implementation	"I did not remember to do everything I had intended"
	Language liability	Misusing or misunderstanding language/environment	"I misused or misunderstood the…language"
	Mismatch between modules	Imperfectly knowing specs, interface; reversed arguments	"I forgot the conventions I had built"
	Reinforcement of robustness	Not handling erroneous input	"I tried to make the code bullet-proof"
	Surprise scenario	Unforeseen interactions in program elements	"forced me to change my original ideas"
	Trivial typos	Incorrect syntax, reference, etc.	"although my original pencil draft was correct"
Eisenstadt [8], industry experts, COBOL, Pascal, Fortran, C *1993*	Clobbered memory bugs	Overwriting memory, subscript out of bounds	Also identified why errors were difficult to find: cause/effect chasm; tools inapplicable; failure did not actually happen; faulty knowledge of specs; "spaghetti" code
	Vendor problems	Buggy compilers, faulty hardware	
	Design logic bugs	Unanticipated case, wrong algorithm	
	Initialization bugs	Erroneous type or initialization of variables	
	Variable bugs	Wrong variable or operator used	
	Lexical bugs	Lexical problem, bad parse, ambiguous syntax	
	Language	Misunderstandings of language semantics	
Panko [18], novice, Excel *1998*	Omission error	"Facts to be put into the model…but are omitted,"	Quantitative errors: "errors that lead to an incorrect, bottom line value"
	Logic error	Incorrect algorithm or incorrectly implemented algorithm	
	Mechanical error	"Typing the wrong number…or pointing to the wrong cell"	
	Overload error	Working memory unable to complete task without error	Qualitative errors: "design errors and other problems that lead to quantitative errors in the future"
	Strong but wrong error	Functional fixedness (a fixed mindset)	
	Translation error	Misreading of specification	

Abbildung 10. Fehlerklassifikationen verschiedener Studien. Aus: Ko & Myers 2003, 8

Ein Aufgabenabhängiges Schema, wie es Pope et al. [1989] und darauf basierend Ebrahimi [1994] vorstellen, empfiehlt sich nicht, da in den vorliegenden Daten verschiedene Aufgaben bearbeitet wurden. Da ihre Kategorisierung in 'malformed', 'missing' 'misplaced' oder 'spurious' auf Plandifferenzen beruht, Pläne jedoch mit der Aufgabenstellung variieren, scheint sie nicht praktikabel. Die von Soloway & Spohrer [1986, in Abbildung 10] benannten Problemklassen hingegen decken einerseits viele der gefundenen Fehler nicht ab und sind andererseits zu spezifisch, so dass diese Fehler in den vorliegenden Daten nicht auftreten (z. B. 'Duplicate Tail-Digit').

Zudem benennen Problemklassen bei Soloway & Spohrer sowohl Ursachen ('human interpreter problem') als auch konzeptionelle Klassen ('Data-Type Inconsistency Problem'). Eine Klassifikation von Fehlern nach Ursachen ermöglicht interessante Einsichten in die Entstehung und damit u. U. auch Maßnahmen zu ihrer Vermeidung. Die erhobenen Daten erlauben jedoch keine Zuschreibung von zugrunde liegenden Fehler-

ursachen [siehe Grams 1990[54], Ko & Myers 2003[55]]. Dazu müssten für jede Aktion – wie im Experiment von Ko & Myers [2003] – die kognitiven Prozesse des Programmierers bspw. durch lautes Denken erfasst werden, um Strategien, Ziele oder fehlerhafte Annnahmen zu ermitteln.

Aufgrund der eingeschränkten Vollständigkeit der aufgezeichneten Sitzungen und der realistischen, unkontrollierten Bedingungen sind eindeutige Hinweise auf Ursachen in der Kommunikation zu selten. Somit kann bspw. nicht eindeutig gesagt werden, ob unzureichendes Verständnis Boole'scher Operatoren, syntaktische Verwechslung zwischen '&&' und '||' oder eine unzureichende Planung für die Entstehung einer fehlerhaften Bedingungsformulierung verantwortlich sind. Daher kann eine Klassifikation nach Ursachen auch in Folge der sich am Quelltext orientierenden Auswertungsmethodik hier nicht vorgenommen werden. Sind Ursachen ersichtlich, werden sie aber benannt.

Auf Programmierexperten zugeschnittene Modelle wie das von Ko & Myers [2003] weisen eine zu geringe Granularität und Aussagekraft zur Beschreibung typischer Anfängerfehler auf. Implementierungsfehler, die Sprachkonstrukte oder Datenstrukturen betreffen, werden bei Ko & Myers [2003] nicht weiter spezifiziert. Ähnlich wird auch bei Pope et al [1989], Ebrahimi [1994] und Johnson [1983, in Abbildung 10] in der Kategorie 'malformed' nicht mehr differenziert, vermutlich weil nur syntaktisch korrekte Anfängerprogramme untersucht worden sind.

Eine der wenigen Studien, die explizit Syntaxfehler einbezieht, haben Hristova et al. [2003] durchgeführt. Die in einer Umfrage ermittelten 20 häufigsten Anfängerfehler in Java wiesen sie den Kategorien Syntax, Semantik und Logik zu, wobei die Einteilung aus Sicht der Anfänger erfolgt ist. Während syntaktische Fehler die Punktuation, Reihenfolge und Schreibweise betreffen, fallen in die Kategorie Semantik Fehler im Zusammenhang mit der Bedeutung des Codes. Diese seien eher javaspezifisch und resultieren möglicherweise aus fehlerhaften Vorstellungen darüber, wie die Sprache bestimmte Anweisungen interpretiert [vgl. Hristova et al. 2003, 154f]. Logische Fehler hingegen werden als generelle, sprachunabhängige Fehler definiert, die sich aber auch in fehlerhafter Syntax und Semantik spiegeln können.

---

[54] Grams [1990, 19] unterscheidet 'Schnitzer' und 'Irrtümer', wobei Irrtümer individuell und überindividuell auftreten können. Letztere stellen nach Grams typische menschliche Denkfallen wie selektive Wahrnehmung (Scheinwerfermodell) dar und können häufige Fehler wie das Außerachtlassen von Ausnahmefällen (z. B. negative Eingabewerte) erklären.

[55] Kognitive Probleme können während allen Aktivitäten im Prozess der Programmentwicklung (z. B. Erzeugen der Spezifikation oder Implementieren eines Sprachkonstruktes) Fehlfunktionen bewirken.

Im Gegensatz dazu werden für das hier verwendete Klassifikationsschema zwar die Kategorien Syntax, Semantik und Logik übernommen, die Zugehörigkeit aber in Anlehnung an die Verwendung der Begriffe in der Softwareentwicklung aus Compiler- bzw. Interpreter-Sicht definiert. Demnach zeigen sich Syntaxfehler bereits zur Kompilier-Zeit an Fehlermeldungen, während semantische Fehler zu Laufzeitfehlern (Exceptions) führen [vgl. Balzert 1999, 172]. Logische Fehler sind aufgabenabhängig und liegen vor, wenn „ein Programm einwandfrei abgearbeitet wird, die Ergebnisse aber nicht oder nur zum Teil mit den erwarteten Ergebnissen übereinstimmen“ [Balzert 1999, 172].

Die Definition der Kategorien anhand von Compiler und Interpreter ermöglicht ein Klassifikationsschema frei von Annahmen über Fehlerursachen und gibt gleichzeitig Hinweise darauf, wie diese Fehler zu entdecken sind. Da der Java-Compiler zusätzliche – eigentlich semantische – Prüfungen bspw. auf Typsicherheit oder das Einbinden der Klassenbibliothek vornimmt, fällt ein Großteil der Fehler in den Bereich Syntax[56]. Obwohl eine engere Definition von Syntax ebenso plausibel ist, orientiert sich Syntax hier an der Spezifikation der Sprache Java und somit am Compiler.

Daher ist jedoch eine weitere Differenzierung der Syntaxfehler sinnvoll, um Schwierigkeiten von Anfängern aussagekräftig beschreiben zu können. Für die Entwicklung der Unterkategorien der Klasse 'Syntax' wurden Ideen aus existierenden Klassifikationen gezogen[57] und die Kategorien den ermittelten Kriterien anpasst. Diese orientieren sich damit weniger an Fehlerursachen als vielmehr konzeptionell an den Sprachkonstrukten und allgemeinen Programmierkonzepten, auf die sich die Fehler beziehen. Da in einer Programmiersprache Konzepte in enger funktionaler Beziehung zueinander stehen bzw. aufeinander aufbauen, sind die Kategorien hier nicht scharf voneinander zu trennen, so dass eine anderweitige Zuordnung einzelner Fehlertypen zu den Klassen denkbar wäre. Das Schema stellt für die Beschreibung der hier gefundenen Fehler jedoch eine hilfreiche Strukturierung dar und ermöglicht eine differenzierte Exploration von Anfängerproblemen.

---

[56] Die Syntax einer Programmiersprache regelt durch eine Grammatik, welche Zeichen als Symbole zulässig sind und wie diese zu Ausdrücken kombiniert werden können [siehe Balzert 1999, 91]. Die Semantik hingegen besagt, wie die Symbole und Ausdrücke zu interpretieren sind. Für Java definiert die Java Language Specification Syntax und Semantik (http://java.sun.com/docs/books/jls/third_edition/html/ j3TOC.html). Nach Schiedermeier [1996] sind „(d)ie Grenzen zwischen Syntax und Semantik (…) verschwommen und lassen sich nicht eindeutig ziehen.“

[57] Ala-Mutka et al. [2005], Spohrer & Soloway [1986], Hristova et al. [2003]

Syntax:

- *Punkt/ Trennzeichen:* Vergessene oder inkorrekt genutzte Kommata, Semikola oder Punkt-Operatoren.
- *Klammersetzung:* Fehlende runde oder geschweifte Klammern, Verwendung der falschen Klammerart. Nicht erfasst werden Klammerfehler, die zu einer logisch falschen Berechung führen.
- *Variablen:* Fehlende oder fehlerhafte Deklarationen und Initialisierungen, auch in Folge von Schreibfehlern oder abweichenden Variablennamen.
- *Datentypen/ Arrays:* Unzulässiges Casten zwischen Datentypen, Methodenaufrufe mit Parametern falschen Datentyps und Inkompatibilität zwischen Rückgabetyp einer Methode und der aufnehmenden Variablen. Sie umfasst Initialisierungsfehler bezüglich der Arraygröße, alle Fehler im Gebrauch der eckigen Klammern und fehlerhafte Zugriffe auf Arraystellen.
- *Klassen/ Methoden:* Hierunter werden alle Fehler eingeordnet, die sich auf die Deklaration oder den Aufruf von Klassen oder Methoden beziehen. Dies betrifft Schlüsselwörter, Groß-/ Kleinschreibung, Parameter, statischen Aufruf ebenso wie die Konvention der Übereinstimmung von Dateiname und Klassenname.
- *Vererbung:* Fehler bezüglich der Vererbung, die sich auf syntaktischer Ebene zeigen, sind fehlende Ableitung von einer Superklasse, fehlende 'import'-Anweisungen oder das Nicht-Implementieren der Methoden eines Interface.

Die ermittelten Fehler werden im Folgenden differenziert nach ihren Fehlerklassen beschrieben und im Hinblick auf ihre Auswirkungen für den Problemlöseprozess analysiert.

4.3 ERGEBNISSE

Die Ergebnisse zu den Häufigkeiten der ermittelten Fehler beruhen auf der Zuordnung der fehlerhaften Code-Fragmente in eine der vorgenannten Klassen. Da Anweisungen gerade bei Anfängern oft multiple Fehler aufweisen, kann eine Anweisung im Gegensatz zu einem spezifischen Fehler wie z. B. ein vergessenes Semikolon in mehrere Kategorien fallen. Die Häufigkeiten sind in Folge der Abhängigkeit von den analysierten Aufgabenstellungen, aber auch im Hinblick auf den Umfang und die Prozesshaftigkeit der Daten als eine Orientierung zu verstehen. Im Sinne der Definition von Problemsituationen lassen sich die Fehlerzahlen in den Kategorien nicht auf die Anzahl der entsprechenden Probleme übertragen. Denn durch die prozesshafte Protokollierung treten möglicherweise aus Flüchtigkeit begangene Fehler gerade bezüglich der Punktuation besonders oft auch. Die in der Darstellung der Ergebnisse aufgeführten fehlerhaften Code-Fragmente können mit den zusätzlichen Informationen zu Entstehungszeit, Identi-

fikationszeit sowie – falls erfolgt – Behebung und diesbezüglicher Kommunikation der Fehlertabelle auf der beiliegenden CD-ROM entnommen werden.

4.3.1 Syntax

Allein 74 % (166 von 224) der ermittelten Fehler verletzen die syntaktischen Konventionen von Java, so dass sie sich zur Kompilier-Zeit an Fehlermeldungen zeigen. Entsprechend des vorgestellten Klassifikationsschemas lassen sich diese Syntaxfehler wie folgt differenzieren: Punktzeichen (19), Klammern (30), Variablen (24), Datentypen/ Arrays (26), Klassen/ Methoden (50), Vererbung (8), Andere (9).

***Punktzeichen*:**

Das Auslassen von Semikola nach einer Anweisung ist hier einer der häufigsten Fehler (14 der 19 Tokens) im Prozess der Programmimplementierung. Ein Semikolon nach einer Schleifendeklaration – möglicherweise durch Übergeneralisierung [Du Boulay 1989, 298] – trat hingegen nur in einem Fall auf[58]. Weitere Fehler stellen ein vergessener Doppelpunkt nach 'case' in einer 'Switch'-Anweisung, der Gebrauch von Kommata als Trennzeichen innerhalb einer 'for'-Schleifen-Deklaration oder ein Semikolon nach dem Update einer 'for'-Schleifen-Deklaration dar. Als Problemsituationen sind diese wiederkehrenden Fehler jedoch nicht anzusehen, da sie in der Regel innerhalb weniger Minuten korrigiert wurden. Oft gingen der Behebung explizite Hinweise von Teammitgliedern ähnlich der folgenden Beispiele (ID 149, 92)[59] voraus.

Entstehungszeit	Code	Korrekturzeit	Kommunikation
11.26.02	`name [i]= Tastatur.leseString()`	11.26.25	[11.26.20: M] ;;;;;;;;;;;"!!!!!!!! [11.26.21: C -> B.] Aber Semikolon dahinter!!
14.00.47	`System.out.println( "Die Zeitspanne beträgt") System.out.println( d, h, m, s)`	14.07.57	[14.05.57: s] da fehlen noch einige ";" nach anweisungen.. [14.06.49: R] wo [14.06.57: s] die letzten beiden zeilen..

[58] Dieses erzeugt zwar keine Fehlermeldung, sondern führt dazu, dass eine Schleife bzw. Bedingung nie oder nur einmal ausgeführt wird [siehe Hristova et al. 2003, 153]. Da der Fehler jedoch im Gebrauch des Semikolons liegt, wird dieser Fehler unter Syntax geführt.

[59] Die ID-Angaben dienen der Identifikation eines fehlerhaften Code-Fragments in der Fehlertabelle auf der CD-ROM.

In nur einer Situation (ID 93) erwiesen sich Korrekturvorschläge anderer Teammitglieder als nicht ausreichend, um eine fehlerhafte Ausgabeanweisung mit durch Kommata getrennten Variablen richtig zu verändern. Nach mehreren Versuchen wurde das Dokument letztlich von einem anderen Teammitglied übernommen und Fehler berichtigt. Viele Fehler wurden aber auch selbst entdeckt und korrigiert, besonders häufig unmittelbar nach dem Kompilieren. Die genaue Benennung dieser Fehler durch den Compiler (';' bzw. ':' expected) scheint demnach eine wirksame Hilfe, wenngleich nicht auszuschließen ist, dass gerade diese eindeutigen Fehlermeldungen Unachtsamkeit bei der Kodierung fördern können: „just letting the compiler do their thinking for them“ [Jadud 2005, 30].

Klammern:

Wie bei Hristova et al. [2003] und Jadud [2005] betrifft ein Großteil der Syntaxfehler die javaspezifische Klammersetzung, besonders für Klassen und Methoden. Obwohl viele der Anfänger die öffnende und schließende Klammer setzen, bevor sie eine Klasse oder Methode mit Anweisungen füllen, gehen über die Hälfte der etwa 30 Klammerfehler auf gänzlich fehlende, falsch positionierte oder nicht schließende geschweifte Klammern bei Klassen, Methoden oder Schleifen zurück.

Der zweithäufigste Fehler mit fünf Tokens ist das Weglassen der runden Klammern bei dem Aufruf einer Methode. Ähnlich der Punktuation konnten diese Klammerfehler jedoch zumeist unmittelbar nach dem Tipp eines Teammitglieds oder nach dem Kompilieren behoben werden. Problemsituationen bezüglich der Klammersetzung scheinen sich dann zu ergeben, wenn weitere Unklarheiten bspw. im Hinblick auf eine Berechnung (ID 23, 25) oder den Einsatz einer Methode für einen speziellen Plan (ID 213) hinzukommen.

Entstehungszeit	**Code**	**Korrekturzeit**	**Kommunikation**
17.12.01	`Name = Tastatur.leseString`		[17.14.27: J. -> tutor1] Also z.B so? Name = Tastatur.leseString …
17.15.15	`Name = Tastatur.leseString[]`	17.19.14	[17.18.35: J. -> tutor1] Wie sieht diese Zeile aus? String name = Tastatur.leseString[] [17.19.01 tutor1] bis auf die eckigen Klammern und das fehlende Semikolon, ein guter Anfang [17.19.03 T] Müssten es nich runde Klammern am Ende sein?

Variablen:

Die meisten Fehler in Bezug auf Variablen entstanden durch Abweichungen des Variablennamens in der Deklaration bzw. Initialisierung und dem späteren Gebrauch (neun Tokens[60]). Dies schließt – vermutlich durch Flüchtigkeit bedingte – Tippfehler, aber auch Unachtsamkeit bei der Variablenbenennung mit ein. Ein Code-Beispiel dafür ist folgendes Fragment (ID 117):

```
int aktuellesJahr = 2005;
 …
if(((jahr%4)==0)&& ((jahr%100)!= 0))
```

Als weitere Fehler waren fehlende Initialisierungen (sieben Tokens) besonders bei den Laufvariablen von 'for'-Schleifen und die Deklaration bzw. Initialisierung zweier Variablen (auch unterschiedlichen Datentyps) mit derselben Bezeichnung zu beobachten (zwei Tokens). Das nachstehende Beispiel zeigt, wie strukturelle Komplexität des Codes – hier durch den Versuch, alles 'zusammenzupacken'– eine fehlerhafte Initialisierung und vergessene Deklarationen begünstigt (ID 80, 81).

Entstehungszeit	Code	Korrekturzeit	Kommunikation
14.37.01	`int bezahlt, fuenfzig, zwanzig, zehn, eins, Preis = (int) (Math.random()*100;`	15.08.01	[14.54.46: s] also das in der ersten zeile passt das mit der aufzählung so nicht ganz.. [14.54.56: s] der preis am ende muss extra aufgeführt werden.. … [14.56.14: S] entweder ihr Deklariert Variablen am Stück, aber dann ohne Wert zuweisung, oder ihr macht alles in eine eigene Zeile ... [15.08.49: s] schreib das bezahlt mal in eine extra zeile und initalisier es mal mit 100;

Ähnlich steht auch in Abbildung 11 die fehlende Initialisierung der Variable 'i' in Verbindung mit der Aufteilung von Funktionalität auf Methoden und Klassen. Wiederum kann aber festgestellt werden, dass mithilfe des Teams bzw. des Compilers derartige Variablenfehler leicht gefunden und behoben worden sind.

[60] Drei der neun Fehler betreffen Schreibfehler bei der Variable 'length' der 'String'-/ 'StringArray'-Klasse.

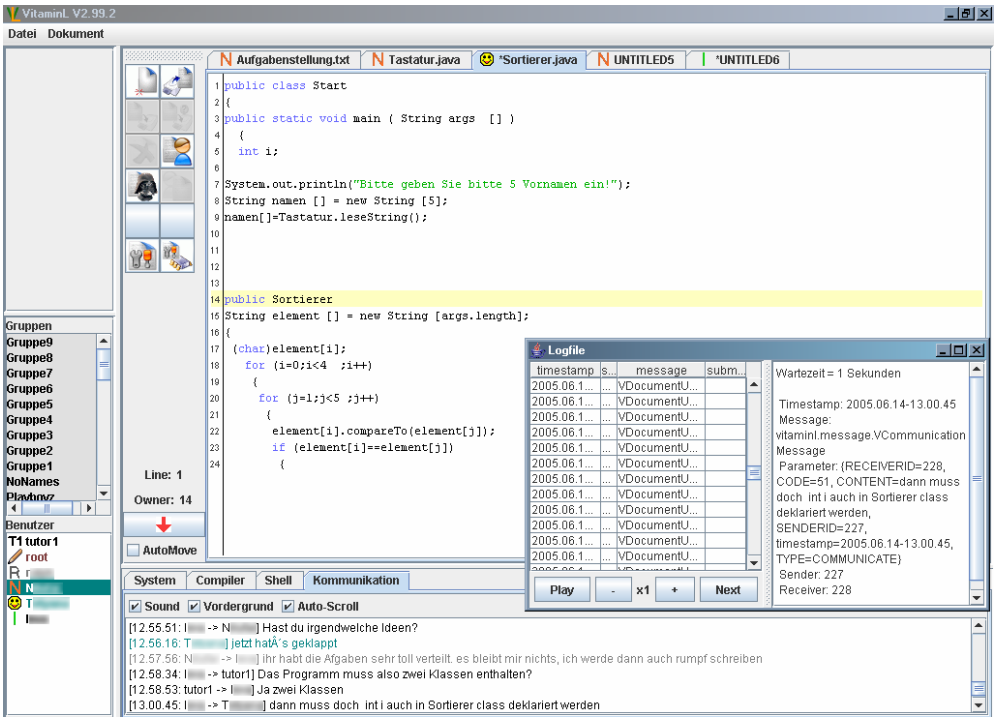

Abbildung 11. Screenshot Benutzertest

Datentypen/ Arrays:

Schwierigkeiten im Umgang mit Variablen treten im Hinblick auf ihren Datentyp auf, besonders wenn es sich um den komplexen Datentyp 'Array' mit seiner eigenen Notation handelt. So betreffen 17 der 26 Fehler in dieser Kategorie die Initialisierung von Arrays und den Zugriff auf einzelne Stellen, vor allem bei der Einbindung von Methoden. Notationsfehler reichen von vergessenen eckigen Klammern, einer fehlenden Längenangabe bei der Initialisierung über die Deklaration bzw. Initialisierung mit runden Klammern oder einer Stellenangabe über eine Laufvariable.

```
public static void main (String[i] args)        (ID 137)

String name[i] = new String[5]                  (ID 217)

string nameone = new String[];                  (ID 153)
```

Während viele dieser Fehler schnell behoben wurden, führten einige erst nach mehreren Änderungen im Sinne einer Strategie von 'Versuch-und-Irrtum' zur Korrektur (ID 137, 8, 165). Das folgende Beispiel (ID 8) verdeutlicht mögliche Probleme bei der Array-Initialisierung.

Entstehungszeit	Code	Korrekturzeit	Kommunikation

11.09.25
 	
11.10.28
	`string[] = namen;`
	
	`string[] namen = {5};`
		11.12.00	[11.08.03: A] mit new irgendwie...hilf mir mal jemand
			[11.08.37: S] String vorname[ ]= new String irgendwie sowas?
			[11.09.05: A] oder reicht die angabe der länge?
			[11.09.32: S] ka
			[11.09.38: M] String name = new String("name")
			[11.10.09: A -> M] ja. wir wollen doch aber gleich ne länge mitgeben, oder?
			[11.10.49: A] was meint ihr dazu?
			[11.11.49: tutor1] String [ ] namen = new String [5];

Als kritisch erwiesen sich Zugriffe auf Arrays, wenn es bspw. in den Benutzertests galt, schrittweise den Array-Stellen Werte zuzuweisen oder diese auszugeben (ID 146/ 147, 162).

Entstehungszeit	Code	Korrekturzeit	Kommunikation

11.19.12	`for (int i=0; i<5; i++)`
	`{`
	`name =`
	`Tastatur.leseString();`
	`}`
	`System.out.println(name);`
		11.21.26	[10.45.34: B] zimbo du array gott, wie geht das mit dem im array spechern nochmal?
			…
			[10.55.19: B -> M] pass auf: wir sind da schon auf nem guten weg... aber wie kriegen wir denn die Eingabe, die er über die Methode Tastatur.leseString einlist in ein array?
			…
			[11.16.49: B.] okay, das ist also der string... toll und die for-schleife, toll... und wie kriegen wir die scheisse in das array???
			[11.19.57: M -> B] was willste denn jetzzt machen
			11.19.20 kompiliert: ! Fehler
			[11.20.18: B -> tutor1] ich sach im, dass er das was er ausliest in das array schreiben soll, aber das geht nicht:-(
			[11.20.36: B] leute, wie gesagt, ich habe keine ahung von arrays, also wärs cool, wenn mir mal wer helfen würde...
			[11.20.37: C] System.out.println("Erster name(i)); name[i] = Tastatur.leseString();

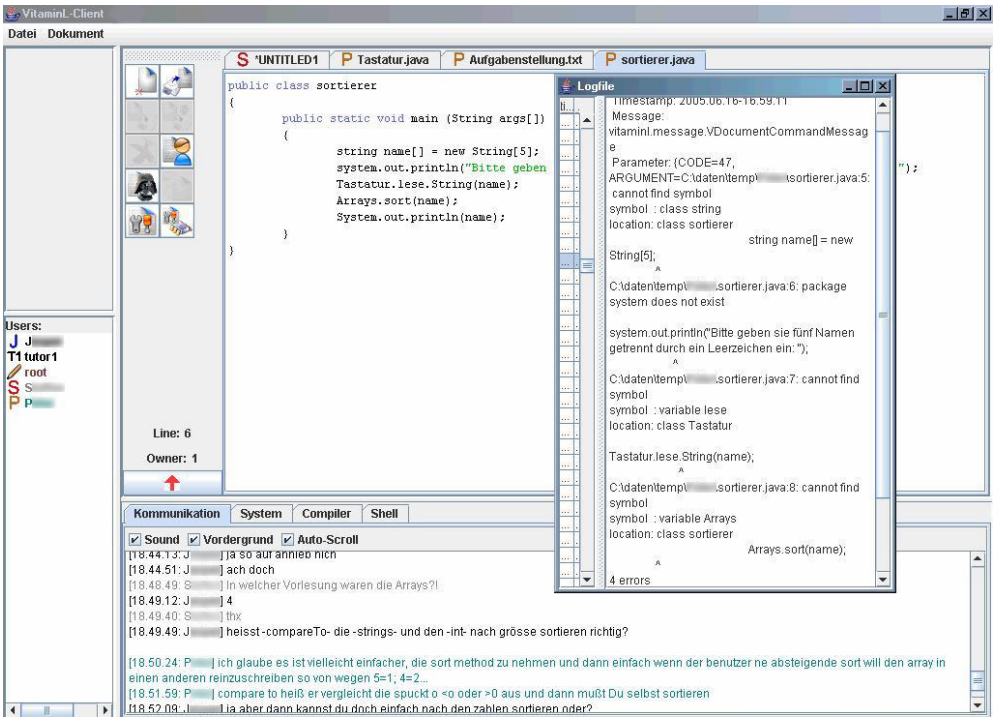

Abbildung 12. Screenshot Benutzertest

Im letzten Beispiel (ID 162) fehlt nicht nur die Angabe der genauen Arraystelle, sondern auch die für das Iterieren über die Stellen nötige 'for'-Schleife. Erst nach etwa einer halben Stunde wurde der eingelesene String korrekt einer Arraystelle zugewiesen, indem für jede der fünf Eingaben eine eigene Einleseanweisung kodiert wurde. Eine adäquate und effiziente Nutzung von Arrays zur Erreichung eines Ziels wurde in diesem Falle dadurch erschwert, dass alle drei Teammitglieder in einem eigenen Dokument versuchten, die Aufgabe zu lösen. So konnten sich die Teilnehmer untereinander kaum bei Fehlern helfen und ihre jeweils wichtigen Ansätze zur Lösung einer Teilaufgabe nicht zusammenführen.

Das Angeben des Array-Namens ohne seine Stellen in einem Methodenaufruf bspw. 'System.out.println(name)' oder 'leseString()' kann auch aus einer allgemeineren Perspektive als ein Problem mit Kompatibilität zwischen Datentypen betrachtet werden[61]. Während die Kommunikation in den beschriebenen Fällen auf die Notation von Arrays als Ursache der Fehler schließen lässt, sind Inkonsistenzen bei Datentypen ebenfalls häufig zu beobachten. So betreffen sechs der 26 Datentypfehler die Inkompatibilität

[61] Durch das Weglassen der Array-Klammern und -stellen wird eine Methode für ein Array, nicht für bspw. die darin gespeicherten Strings aufgerufen. Semantische Missverständnisse oder Notationsschwierigkeiten könnten Ursachen sein.

zwischen dem Rückgabewert einer Methode und der aufnehmenden Variablen ähnlich der folgenden Situation (ID 10).

Entstehungszeit	Code	Korrekturzeit	Kommunikation

10.47.01	`string eingabe;`
	
	`eingabe = Tastatur.leseInt();`
	…
		11.24.03	[10.37.29: A. -> S.] ich such mal in einem alten programm
			[10.40.50: S.] war das nicht was mit leseint.tastatur?
			[10.41.05: A.] eingabe=Tastatur.leseInt();eingabe=Tastatur.leseInt();
			…
			[11.23.20: tutor1] wie wäre es mit leseString(), die gibts in Tastatur nämlich auch
			[11.23.57: S] aha! stimmt...das war ja immer nur das BSP mit den zufallszahlen

Diese mangelnde Differenzierung, wie hier aufgrund früherer Programmierepisoden, trat auch in umgekehrter Richtung als '`int name = lese.String();`' (ID 132) auf. Weitere vereinzelte Fehler waren unzulässiges Casten von 'String' nach 'Char' und Inkompatibilität zwischen deklariertem Rückgabetyp und tatsächlichem Rückgabetyp einer Methode. In der Regel konnten diese Datentyp- und Arrayfehler im Team gemeinsam schnell korrigiert werden. Falls nicht, deutet die Kommunikation darauf hin, dass generelle Schwierigkeiten in der Planung des Programms z. B. dem Aufruf oder der Erstellung von Methoden zusammenwirkten.

Methoden/ Klassen

Wie in einigen Code-Beispielen mit multiplen Fehlern bereits sichtbar geworden ist, traten Fehler bei Methodenaufrufen nicht nur bezüglich des Datentyps auf. Ein weiterer häufiger Fehler war der Aufruf einer Methode mit fehlerhaftem Bezeichner (sieben Tokens). Diese reichten von Tippfehlern, wie dem Fehlen eines Buchstabens, bis zu lediglich ähnlichen Methodennamen.

`int name = tastatur.String();` (ID 131) *korrekt: Tastatur.leseString();*

Ebenso wie in Aufrufen von Methoden anderer Klassen ohne ein Objekt oder den Klassennamen bei statischen Methoden (drei Tokens) scheinen sich hier auch Schwierigkeiten mit dem systematischen Einsatz von Methoden und ihren Parametern zu manifestieren.

Entstehungszeit	Code	Korrekturzeit	Kommunikation

16.56.10
 	
17.03.23
 	
17.05.05
	`Tastatur.lese.String(name);`
	
	`Tastatur.leseString(name);`
	
	`Tastatur.leseString();`
	`name[]=leseString();`
		17.34.01	[17.02.57: J] und das funzt?
			[17.03.10: J] jetzt so
			[17.03.11: P] noch nicht
			...
			[17.15.25: J -> P] bekommst du in deinem porg die ausgabe der namen hin also eionfach ur mal von dem was du eingegeben hast ich nich
			...
			[17.16.29: P] nee ich komm grad mit der tastatur klasse nicht klasse irgenwas is mit der variable falsch
			…
			[17.32.47: P] bei mir kommt trotzdem immer die fehlermeldung cannot find method leseStrin()
			[17.33.36: tutor1 -> P] ja der Fehler ist ja noch da. es gibt nur Tastatur.leseString()

Obwohl mithilfe des Compilers die ersten Fehler schnell gefunden wurden, stellte der Transfer zwischen den beiden letzten Zeilen ein tatsächliches Problem dar, dass nur nach der expliziten Lösung durch den Tutor überwunden werden konnte. In anderen Fällen zeigten sich die Aufrufschwierigkeiten in dem Versuch, die Tastaturklasse zu importieren (ID 39), der Deklaration der Tastaturklasse (ID 84) oder der Deklaration einer ihrer Methoden in der eigenen Klasse (ID 212). Hier konnten die Teammitglieder jedoch schnell helfen.

Entstehungszeit	Code	Korrekturzeit	Kommunikation

13.20.06
 	
ID 84
	`class aufgabe13`
	`{`
	`public static void main(`
	`String  [] args )`
	`{`
	`    int sekunden;`
	`}`
	`public class Tastatur`
	`}`
		13.24.15	[13.19.54: s] nicht klar?
			[13.20.09: R] mir irgendwie auch ncit
			[13.20.49: s] also: eine Klasse stellt ja neben Variablen auch methoden zur verfügung..klar soweit?
			[13.21.25: C] ok
			…
			[13.22.04: s] und genauso stellt die klasse tastatur die methode leseInt bereit..
			[13.23.09: R] wir müssen also leseint einbauen
			[13.23.29: s] genau..

Auch weitere Fehler wie eine fehlerhafte Parameterübergabe ('`nummer1.setBaujahr =2003;`') (ID 16, 18), die Erstellung zweier Konstruktoren mit denselben Parametern (ID 60), eine fehlende Angabe des Rückgabetyps einer Methode (ID 26) und das Erzeugen von Methoden innerhalb der statischen 'Main'-Methode (ID 17, 19) stellten keine Problemsituation für das Team dar.

Der Großteil der – ebenfalls unproblematischen – Fehler in dieser Kategorie betraf die Groß-/ Kleinschreibung von Klassennamen (zehn), besonders der Klassen 'String' (sieben) und 'System'. Unachtsamkeit oder Übergeneralisierung der Kleinschreibung von primitiven Datentypen wie 'int' wären denkbare Ursachen. Fast immer wurden diese Fehler sofort nach dem Kompilieren oder nach einem Tipp korrigiert. Gleiches gilt auch für die zehn Fehler aufgrund eines Klassennamens, der – oft durch Abweichungen in der Groß-/ Kleinschreibung – nicht mit dem Dateinamen übereinstimmte.

Vererbung:

Als letzte Fehlerklasse, die sich zu Compile-Zeit zeigt, wurden Fehler im Hinblick auf das Nutzen von Methoden einer Superklasse oder Schnittstelle beobachtet. Die sieben Fehler umfassen ein fehlendes 'extends' zur Ableitung, eine fehlende oder fehlerhafte 'import'-Anweisung oder das Erzeugen von Objekten einer abstrakten Schnittstelle. Die geringe Anzahl an analysierten Aufgaben zur Objektorientierung und Vererbung dürfte gerade die Fehlerhäufigkeit in dieser Kategorie beeinflusst haben. Während in den drei Fällen eines fehlenden 'imports' oder 'extends' das Kompilieren direkt zur Lösung führte, kann für die anderen Fehler keine Aussage getroffen werden. Denn sie entstammen Logfiles, die während der Vorlesung mit anschließender Übung entstanden und daher keine verlässlichen Indikatoren bieten. Eine Auswertung der Logfiles zu späteren Sitzungen der virtuellen Veranstaltung wäre hier aufschlussreich.

4.3.2 Semantik

Anhand der Daten wurden elf Fehler ermittelt, die in der Regel eine Exception zur Laufzeit bedingen. Diese Fehler lassen sich in zwei Gruppen unterteilen: Das Nichtspeichern bzw. Nichtverwenden der Rückgabe einer Methode (fünf) und Zugriffe auf eine nicht vorhandene Array-Stelle (sechs), zumeist durch falsche Werte für die Laufweite einer Schleife.

Während ein Großteil der Fehler im Zusammenhang mit dem zielgerichteten Aufruf von Methoden sich bereits beim Kompilieren zeigt, lassen sich Fehler bezüglich der Verwendung von zurückgegebenen Werten erst zur Laufzeit feststellen. In den hier betrachteten Fällen sind die Folgen bei der Ausführung des Programms zumeist 'NullPointerExceptions', da im Verlauf auf ein leeres Array zugegriffen würde. Diese Fehler traten zumeist bei der Nutzung der Methoden der Klasse 'Tastatur' zum Einlesen

von Konsoleneingaben auf. Auch im nächsten Ausschnitt wird 'leseString( )' aufgerufen als handele es sich um eine Anweisung und nicht um eine Methode mit Rückgabe. Derartige Fehler entsprechen dem von Bonar & Solway [1989, 333] beschriebenen 'Implizitlassen' nötiger Anweisungen, möglicherweise durch undifferenzierten Transfer aus der natürlichen Sprache.

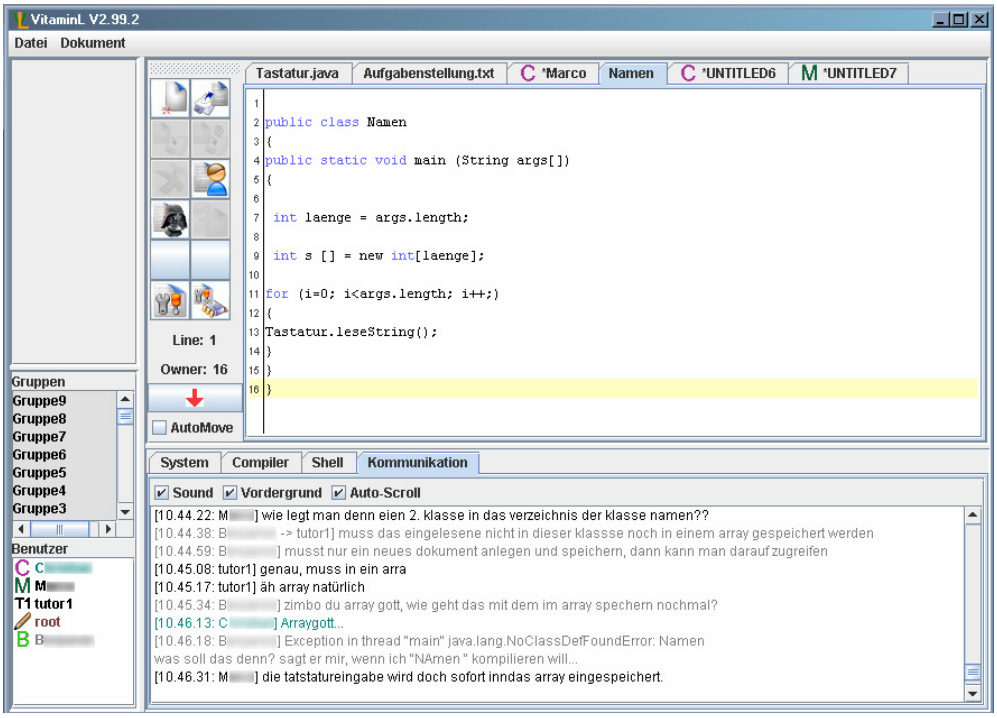

Abbildung 13. Screenshot Benutzertest

Erst etwa 25 Minuten später wird der Quelltext so – syntaktisch fehlerhaft – abgeändert, dass das Array die Rückgabe aufnimmt. Die Problemsituation erwächst hier aus einem Kommunikationsproblem. Denn die Nachfrage an den Tutor (Abbildung 5: Chat: 10:44:38) gibt Hinweise darauf, dass der Fehler erkannt wurde. Jedoch scheint die Nachricht des Teammitglieds, dass die Tastatureingabe bereits im Array gespeichert wird (Abbildung 5: Chat: 10:46:31), das Verfolgen des richtigen Lösungsansatzes zu behindern. Es ist anzunehmen, dass hier ein Problem mit deiktischen Verweisen im virtuellen Raum [Göldner 2005] vorliegt, da sich diese Kommunikationsaktion vermutlich auf ein anderes Dokument, nämlich das des betreffenden Teammitglieds, bezieht.

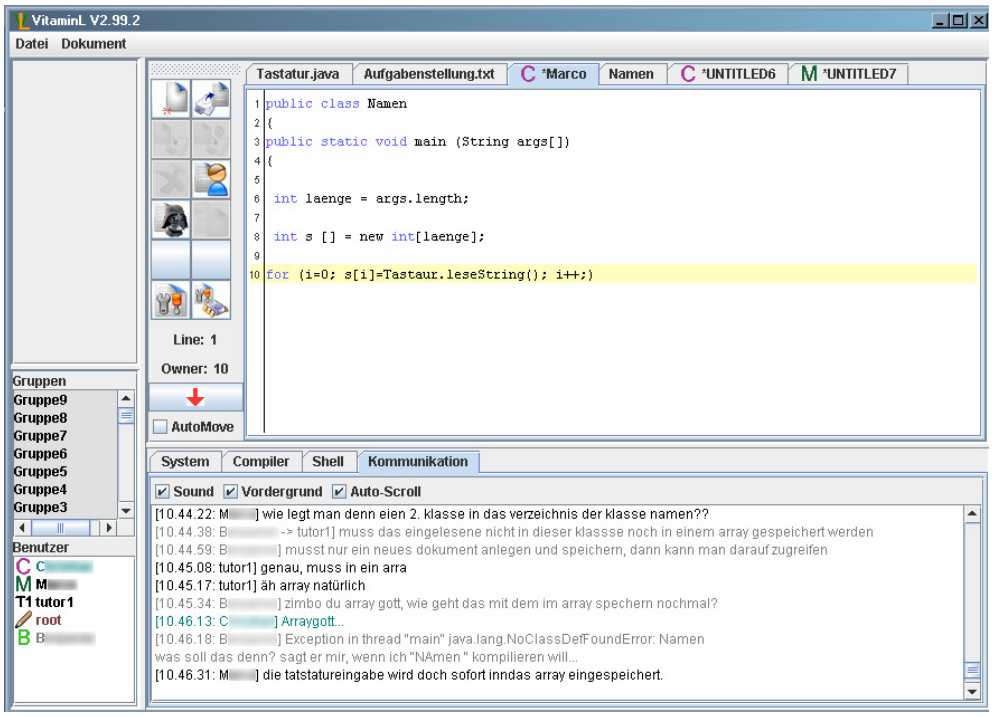

Abbildung 14. Screenshot Benutzertest

In anderen Fällen liegen zum weiteren Umgang mit diesem Fehler keine Informationen vor oder er wurde durch erfolgreichen Wissenstransfer behoben.

Der zweite Fehlertyp in dieser Kategorie, die 'ArrayIndexOutOfBoundsException', trat zumeist in Folge einer direkten Übernahme der Array-Länge als absolute Zahl auf (ID 191, 201, 145).

```
String element = new String [5];
…
for (i=0;i<5 ;i++)
 {
  for (j=1;j<5 ;j++)
   {
```

In nahezu allen Situationen wurden diese Fehler im Laufe der weiteren Implementierung ohne Auszuführen selbst entdeckt.

Auch im folgenden Beispiel, in dem die Stellenangabe in Zeile 79 zusammen mit der Laufweite der Schleife die 'ArrrayIndexOutOfBounds-Exception' verursacht, konnte der Fehler – diesmal durch die Hilfe eines Teammitglieds – korrigiert werden. Hier übernimmt ein Teammitglied gerade die berichtigende Bearbeitung des Dokuments, das kurz vorher nach dem Auftreten der Exception von dem bis dahin implementierenden Mitglied freigegeben worden war.

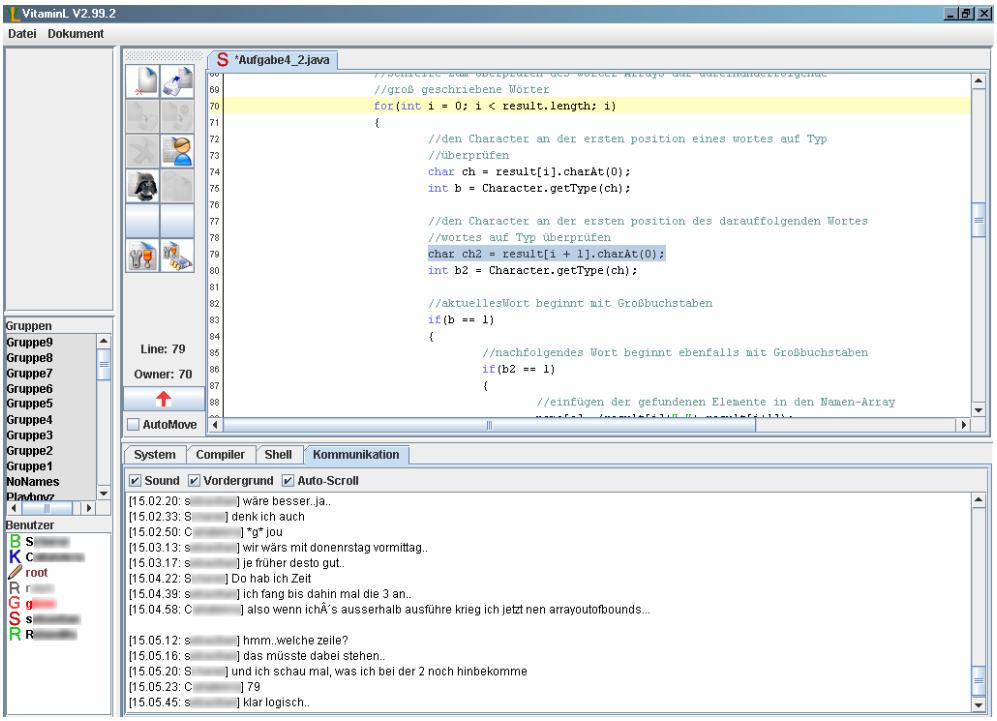

Abbildung 15. Screenshot Benutzertest

Der Problemlöseprozess konnte so zwar ohne Stocken fortgesetzt werden, jedoch ist unklar, ob die Ursache der Ausnahme für alle anderen Teilnehmer auch offensichtlich war. Der Fehler wurde behoben, ohne über die fehlerhafte Code-Stelle zu kommunizieren. Während die Fehlermeldung zur Laufzeit hier erfolgreich als Hinweis auf den Fehler genutzt wurde, sind logische Fehler nur aus nicht den Erwartungen entsprechenden Programmausgaben abzuleiten.

4.3.3 Logik

Unter den 33 Fehlern in dieser Kategorie finden sich rein aufgabenspezifische Fehler, aber auch übertragbare – generelle Fehlermuster. Zu ersteren sind fehlende oder falsch positionierte Ausgabeanweisungen und mathematische Berechnungen (zwei Tokens) – z. B. aufgrund einer unpassenden Klammerung oder des Einsatzes eines Modulo-Operators – zu zählen. Auch fehlerhafte Startwerte für Laufvariablen einer Schleife, die zur Lösung einer bestimmten Aufgabe nicht analog zum schrittweisen Durchlaufen eines Arrays bei null beginnen, sind hier beobachtet worden (drei Tokens). Einzelne Fehler waren das Überschreiben von Variablenwerten beim Tauschen von Array-Positionen oder die Überdeckung von Variablenwerten, aus welcher eine nie eintretende Bedingung resultierte.

Als Fehler wurden ebenfalls nicht berücksichtigte Werte bzw. Fälle gewertet. In zwei Lösungsversuchen (ID 118, 96) wurden mögliche negative Eingaben des Benutzers, die in der entsprechenden Tastaturmethode zur Rückgabe eines Standardwertes (-1) und in Folge zu einer falschen Berechnung führen würden, nicht in Erwägung gezogen. In der Regel wurden solche potentiellen Ausnahmewerte jedoch durch Prüfbedingungen in Verzweigungen oder Schleifen abgefangen. Bei der logisch und semantisch korrekten Formulierung derartiger Bedingungen traten auch einige Fehler auf. Obwohl vereinzelt Fehler allein bezüglich der Verknüpfung von Operatoren nach Boole'scher Logik begangen wurden, hingen all diese Fehler stark mit der Programmkomposition allgemein zusammen.

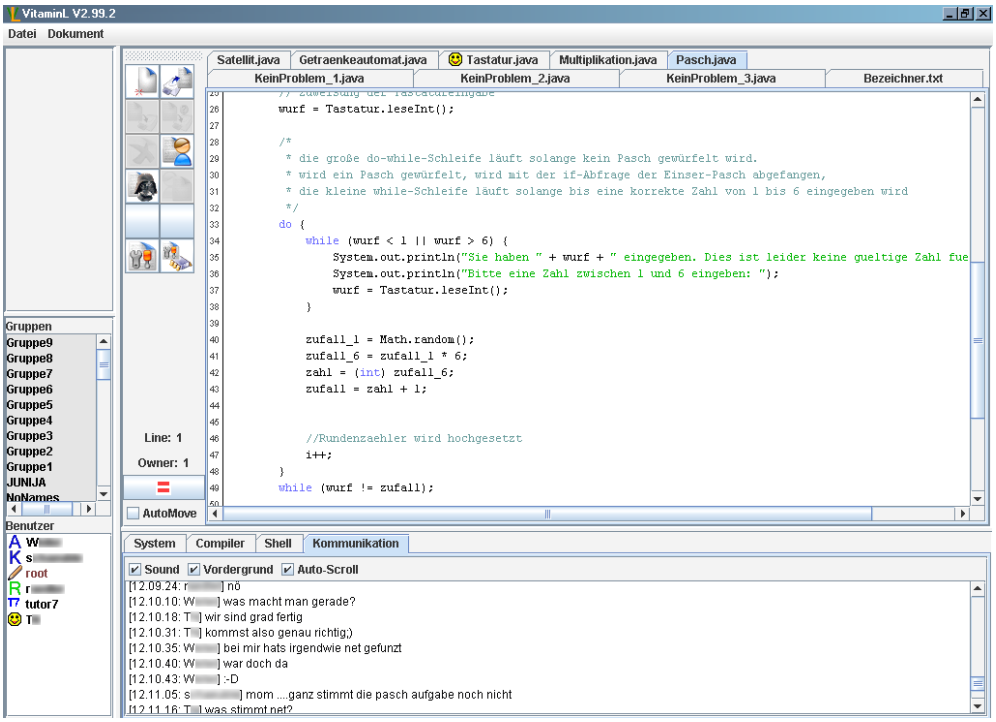

Abbildung 16. Screenshot Benutzertest

Da in diesem Beispiel (ID 121) innerhalb der Schleife keine erneute Benutzereingabe erfolgt, wird die 'while'-Bedingung ausgeführt bis der Computer eine Zufallszahl generieren kann, die der Eingabe entspricht. Diese fehlende Tastatureingabe wurde auch in einem anderen Fall (ID 21) beobachtet. Nach der Feststellung eines Fehlers im Programm fügte ein Teilnehmer unmittelbar die entsprechende Eingabeanweisung ein, ohne seine Lösung jedoch zu kommunizieren. Im weiteren Verlauf dieser Aufgabenlösung kommt es zu einigen Fehlern bei Bedingungsformulierungen ('while' wird nie ausgeführt, '=' statt '==', fehlende Teilbedingung), die sich alle durch erfolgreichen Wissensaustausch schnell korrigieren lassen.

Als letzter Fehler soll die Nutzung von 'zweifelhaften' Plänen in einem Programm beschrieben werden. In zwei der fünf Benutzertests sowie in einer Aufgabe im Rahmen der virtuellen Veranstaltung wurde zur Eingabe von Werten über die Konsole zusätzlich zu der jeweiligen Methode der Tastaturklasse mit der Parameterübergabe durch 'args[]' der 'main-Methode' gearbeitet.

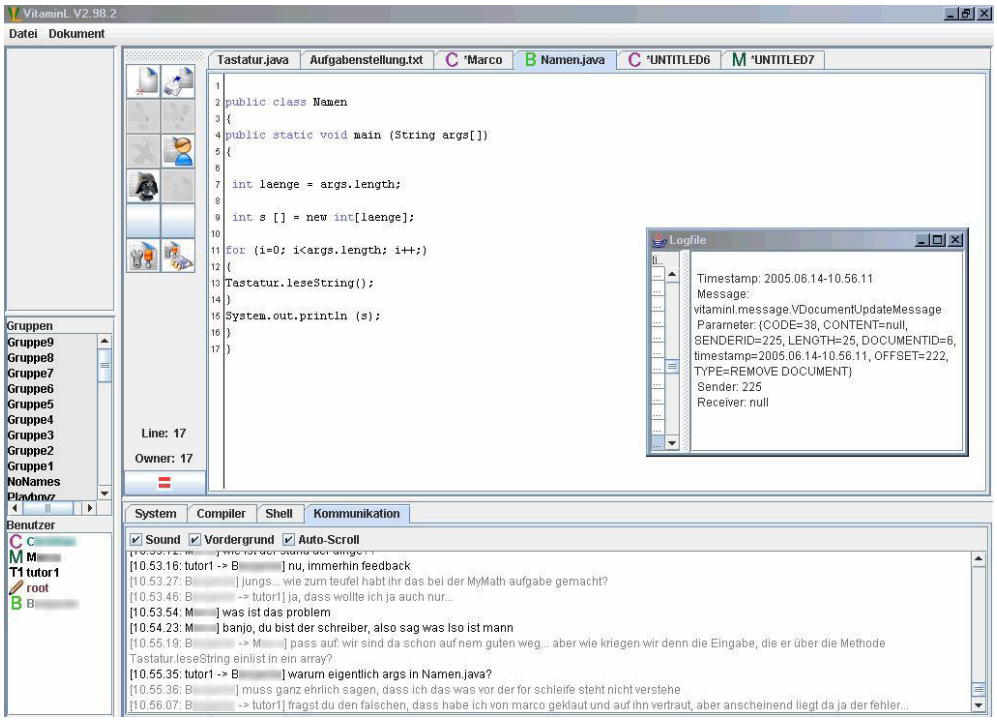

Abbildung 17. Screenshot Benutzertest

Wie in diesem Beispiel wurde auch bei einer anderen Gruppe die Array-Größe für die Tastatureingaben in Abhängigkeit von der Länge der Übergabeparameter definiert. Im besten Falle entsteht dadurch ein korrektes Programm, das von dem Benutzer mehrmals die Eingabe von fünf Namen verlangt. Hier allerdings kann es zu einer nie ausgeführten Schleife führen, da der Benutzer auch keine Parameter bei Programmaufruf übergeben könnte. Die Kommunikation deutet dabei nicht auf semantische Missverständnisse hin, sondern auf den Einfluss früherer Programmierbeispiele bei der Überwindung von Schwierigkeiten, die einzelnen Komponenten zusammenzusetzen.

Entstehungszeit	**Code**	**Korrekturzeit**	**Kommunikation**

10.21.16	`int laenge = args.length;`
	`int s [] = new int[laenge];`
	
	`for (i=0; s[i]=; i++;)`
		11.14.17	[10.24.55: C] Ey M ist das der Quellcode von der Matheaufgabe??
			…
			[10.27.54: B] müsste doch genauso gehen wie bei der MyMath aufgabe, nur mit nem string halt...
			…
			[10.59.42: B. -> tutor1] args.length ist

ID 130			gleich der anzahl der argumente der eingabe oder nicht? [11.00.01: M.] das array trägt genauso viel args wie eingegben … [11.00.33: M -> tutor1] ist das der richtige ansatz ode rsollen wir alles löschen? … [11.13.11: B -> M] und was heisst das über der for schleife... int laenge = args.length... macht das überhaupt sinn?

Gerade in logischen Fehlern scheinen sich die in Kapitel 2 berichteten Anfängerprobleme der 'Übersetzung' allgemeinsprachlicher oder semi-algorithmischer Ziele in miteinander interagierende Programmpläne zu manifestieren.

4.3.4 Zusammenfassung von Problemsituationen

Syntaktische Fehler stellen den überwiegenden Teil der anhand der vorliegenden Daten feststellbaren Fehler dar. In Übereinstimmung mit Hristova et al. [2003] und Jadud [2005] sind Klammersetzungs- und Punktuationsfehler in Anfängerprogrammen sehr häufig. Auch Fehler in der Kompatibilität von Datentypen [Soloway & Spohrer 1986a, b], gerade im Hinblick auf die Rückgaben von Methoden sowie bezüglich Methoden und Parametern generell, konnten festgestellt werden [Hristova et al. 2003]. Fehler in der Deklaration bzw. Initialisierung von Variablen kamen ebenso vor, wobei nicht ersichtlich ist, ob Misskonzeptionen [Samurcay 1989; Du Boulay 1989] oder Flüchtigkeit ursächlich dafür waren. Während bspw. bei Hristova et al. [2003] erwähnte Fehler im Umgang mit Boole'schen Operatoren, '=' vs. '==', 'equals' vs. '==' hier – sicher auch aufgabenbedingt – kaum beobachtet wurden, waren gerade Fehler in der Nutzung von Arrays zahlreich. Ebenfalls aufgrund der analysierten Aufgaben nicht aussagekräftig zu ermitteln sind spezifisch objektorientierte Fehlerquellen.

Wie in einigen Beispielen erkennbar wird, spiegeln sich in den vielen syntaktischen Fehlern häufig übergeordnete Probleme mit der Zerlegung von Funktionalität für einen Programmentwurf und der Übersetzung von Zielen in Pläne einer Programmiersprache. Im Einklang mit den in Kapitel 2.3 vorgestellten Befunden zu typischen Anfängerproblemen deutet besonders die Kommunikation vor und während der Implementierung auf diese Schwierigkeiten der Komposition hin. Da Fehler verschiedene Ursachen haben können, sind diese Probleme am Code nicht direkt nachzuweisen. Einige Kom-

munikationsausschnitte aus verschiedenen Benutzertests seien aber exemplarisch genannt.

> [16.43.10: S] Man müsste wohl einen Array anlegen, der die compareTo(String)-Methode benutzt und dann wieder von vorn bis hinten durchlaufen wird, um den Spaß zu sortieren. Das kann ich aber irgendwie nicht so wirklich...HELP
> …
> [17.42.05: S] Du bist gut...ich hab gerade keine Ahnung wie ich die compareTo-Methode einbauen kann...Aber ihr müsst die Eingabe durchlaufen mit einer for Schleife...

> [13.56.36: Sch] ist der Tutor denn schon da?
> [13.56.47: C] gerade gekommen
> …
> [13.57.09: s] ich hab nämlich keine ahnung wie das gehen soll..
> [13.57.19: C] hehe
> [13.57.29: R] A jetzt bist du am zug
> [13.59.28: C] ok, lad mal grad jemand den geheimtext rein!!
> [14.02.26: Sch] soll man das jetzt einlesen und jeden möglichen Shift von -10 bis +10 ausgeben, oder wie soll das gehen?
> [14.03.20: C] ok, also anscheinend die einzelnen wörter als string in nem array speichern, entschlüsseln und dann nur nohc zählen wie oft die einzelnen wörter vorkommen
> [14.03.38: s] okay..ich fang jetzt mal..
> [14.04.01: C] ok.....und es reicht wenn wir von -3 bis +3 ausprobieren
> [14.08.15: s] okay..
> [14.08.17: s] und nu??
> [14.11.45: s -> C] ich weiss nimmer weiter..:)
> [14.12.16: C] ok moment.....
> [14.15.38: C] also im prinzip funktionierts jetzt wir in dem zweiten beispiel für die dekodierung....

> [10.26.37: B] jungs, ich bin nicht in der lage online zu gehen, somit völlig aufgeschmissen, mache mal einen auf ideengeber: ich würde sagen erstmal die Eingabe über die Tastaturklasse, das dann in nem array speichern und irgendwie über diese comparTo Methode systematisieren

Problemsituationen lassen sich damit nicht allein am Code festmachen. Vielmehr können lange Intervalle ohne Aktionen im Quelltext auf derartige Entwurfsprobleme hinweisen. In anderen Fällen ist in der Kommunikation außer Verweisen auf frühere ähnliche Aufgaben kaum Programmplanung enthalten bzw. wird sich jedem Teilziel einzeln genähert.

> [16.33.42: S] tja, wie fangen wir da bloß an????
> [16.34.21: H] Zunächst mal brauchen wir eine Tastatur eingabe
> [16.34.46: S] unser liebling!
> [16.34.54: H] Eine Ausgabe... Ich schreibe mal ein paar Sachen auf, diemir spontan einfallen
> [16.35.26: S] wisst ihr noch, wie das genau ging mit dieser tastatur?

Dies entspricht theoretischen Forschungsergebnissen, nach denen sich Anfänger in der Regel in einem opportunistischen Prozess zeilenweise dem Ziel nähern, ohne in die Breite ein Programm explizit zu planen (vgl. Kap. 2.2). Konfrontiert mit einer solchen Problemsituation, in der bspw. nicht klar ist, wie mithilfe einer Methode ein Array nacheinander in einer 'for'-Schleife mit Werten belegt werden kann, stellten Hancock et al. [1989] verschiedene Problemlösetypen fest. Während 'movers' sich durch das Programm 'kämpfen' bzw. 'raten' und dabei u. U. diverse Syntaxfehler begehen, neigen 'stopper' dazu, zeitnah um Hilfe zu bitten. Beide Verhaltensweisen ließen sich in den Benutzertests beobachten. In einigen Tests stimmten Teilnehmer jeden Einzelschritt mit dem Tutor ab bzw. ließen sich den aktuellen Ansatz bestätigen. 'Mover' hingegen versuchten zumeist unter Zuhilfenahme von Beispiel-Code früherer Aufgaben ein Programm zu implementieren und später mithilfe des Debuggers oder des Teams bzw. Tutors die Fehler zu beseitigen. Ähnlich der Ergebnisse von Soloway & Spohrer [1986a, b] werden syntaktische oder semantische Fehler demnach auch aufgrund fehlenden Wissens über Programmierpläne und algorithmisches Problemlösen eingeführt.

Für diese syntaktischen Fehler erwiesen sich die umfangreichen Prüfungen des Compilers und seine Fehlermeldungen zumeist als hilfreich, da sie oft unmittelbar nach dem Kompilieren korrigiert wurden. Das Hinzuziehen von Beispielprogrammen war eine häufig genutzte Problemlösestrategie, da aus ihnen nicht nur Syntax und Semantik, sondern auch konkrete Ansätze zu Komposition eines Programms zur Erreichung eines Ziels übertragen werden können. Beim Einsatz dieser prinzipiell effektiven Lern- und Problemlösestrategie wurden jedoch nicht immer geeignete Beispiele ausgewählt oder die tatsächlichen strukturellen Ähnlichkeiten herausgefiltert [siehe Kap. 2.2]. Fast alle Datentypfehler und die überflüssige bzw. fehlerhafte Verwendung der Parametereingabe der 'Main-Methode' deuten auf einen 'Receny-Effect' einer Aufgabe, in der aus Integer-Werten der Durchschnitt errechnet werden sollte. Obwohl strukturelle Eigenschaften dieser Aufgabe übertragbar sind (Iterierende Array-Zuweisung) hielten die Teilnehmer hier in zwei Fällen gerade an den nicht übertragbaren Plänen der Parameterübergabe fest.

Wie die Beispiele gezeigt haben, liegt die Besonderheit in dieser Auswertung in der Erfassung der Prozesshaftigkeit der Implementierung. Dabei werden die Vorteile des kollaborativen Programmierenlernens deutlich. Durch Wissenstransfer und Rollenverteilung – ähnlich dem Pair Programming– können viele Fehler schneller erkannt und behoben werden [vgl. Williams et al. 2002, 206]. Um diese Synergieeffekte auszunutzen

und möglicherweise Wissenslücken des anderen ausgleichen zu können, ist eine effektive Zusammenarbeit nötig. Bei der ersten Nutzung der virtuellen Arbeitsform konnten viele Problemsituationen einzelner Mitglieder aufgrund mangelnder Kommunikation und Zusammenarbeit nicht überwunden werden. Nahezu in allen Tests begannen die Teammitglieder mit der Implementierung in einem eigenen Dokument, wobei sie oft die gleichen Schwierigkeiten hatten. Auch das Kopieren fehlerhafter Code-Fragmente von einem Teammitglied trat wiederholt auf. Zusätzlich mögliche Kommunikationsprobleme, wie sie Göldner [2005] für die virtuelle Programmierung von Anfängern beschreibt, können fachliche Problemlagen verstärken bzw. ihre Überwindung erschweren.

Die vorgestellten Unterstützungsmöglichkeiten werden nun daraufhin untersucht, welche Potentiale sie im Hinblick auf die soeben modellierten fachlichen Problemsituationen von Programmieranfängern in virtuellen Teams bieten. Aus der Diskussion der Techniken folgend wird dann ein möglicher Ansatz zur Integration in die VitaminL-Plattform skizziert.

5 ANSÄTZE ZUR AUTOMATISCHEN UNTERSTÜTZUNG IN PROBLEMSITUATIONEN

Auf Basis der erhobenen Daten werden nun die in Kapitel 3 vorgestellten Analyse- und Unterstützungsmöglichkeiten auf ihre Eignung zur Integration in die kollaborative Lernplattform VitaminL geprüft. Daraus resultierend wird der Ansatz des beispielbasierten Programmierens aufgegriffen und in einer prototypischen Implementierung in Benutzertests und Benutzerbefragungen getestet. Anschließend folgt die Vorstellung maschineller Lernverfahren, anhand derer die mögliche Weiterentwicklung des beispielbasierten Systems experimentell aufgezeigt werden soll.

5.1 DISKUSSION MÖGLICHER ANSÄTZE

Die in Kapitel 3 vorgestellten Techniken zur Unterstützung des Programmierprozesses und zur automatischen Fehlerfindung stellen unterschiedliche Ansätze für die Lösung von Problemsituationen von Anfängern dar. Alle diese Techniken können bei bestimmten Problemen und in spezifischen Phasen – zumeist im Debugging – sinnvolle Hilfestellungen bieten. Im Hinblick auf die Ergebnisse der vorliegenden empirischen Untersuchung scheinen einige Verfahren für die Integration in VitaminL jedoch einen höheren Mehrwert zu versprechen.

In Anbetracht der häufig beobachteten Syntaxfehler von Anfängern könnten visuelle Hervorhebungen zusammengehöriger Klammern oder syntaktisch fehlerhafter Code-Fragmente im Quelltext ein frühzeitiges Beheben solcher Fehler fördern. Indem multiple Syntaxfehler in einem Programm dadurch vermutlich eingeschränkt werden, wären mehr kognitive Ressourcen für das Planen und Debuggen der Semantik und Pragmatik eines Programms frei. Vorstellbar wäre in Anlehnung an Eclipse der Einsatz eines inkrementellen Compilers, der den Quell-Code im Hintergrund kompiliert und die fehlerhaften Zeilen im Editor markiert.

Zusätzlich erweiterte Fehlermeldungen zur Einschränkung der Fehlerursachen von Compiler-Meldungen könnten durch Nennung aller potentiellen Ursachen einer bestimmten Compiler-Meldung (wie BlueJ Kap. 3.1) oder mithilfe eines Klassifikationsverfahrens nach Fox & Uti [2003] (Kap. 3.2.1) realisiert werden. Das automatische Debuggen von Syntaxfehlern durch String-Matching zu häufigen Fehlermustern [Hristova et al. 2003, siehe Kap. 3.2.1] oder ein 'Parser-Repair' wie bei Franek & Sykes

[2004] (Kap. 3.2.1) dürften den Anfängern gerade zu Beginn des Lernprozesses hilfreich sein.

Die hier analysierten Daten lassen jedoch darauf schließen, dass Syntaxfehler zwar zahlreich auftreten, ab einem gewissen Kenntnisstand in der Regel aber mithilfe der Standardmeldungen des Compilers zu beheben sind.

Techniken zum automatischen Debuggen von semantischen oder logischen Fehlern setzen hingegen die syntaktische Korrektheit des Codes voraus. Das Automatisieren von Testfällen kann logische Fehler aufzeigen, jedoch erfordert dies die manuelle Erstellung und Annotation dieser Testfälle für jedes Programm. Vor allem stellt dies aber erst für einen vollständigen Lösungsversuch eine sinnvolle Hilfestellung dar, da Tests selbst für syntaktisch korrekte Teillösungen aufgrund fehlender Funktionen fehlschlagen. Auch bei der statischen Strukturanalyse in ELP wird über die abstrakte Beschreibung der Struktur einer Musterlösung 'loop-assignment' hinaus keine weitere Implementierungshilfe gestellt. Im Hinblick auf die Schwierigkeiten einen semi-algorithmischen Programmablauf in konkreten Code umzusetzen, scheint diese Unterstützung allein daher nicht ausreichend. Auch Visualisierungstools bieten keine Hinweise auf Fehlerursachen; sie konzentrieren sich auf die Veranschaulichung von Datenfluss und Kontrollfluss in Programmen. Empirische Ergebnisse zu ihrer Nützlichkeit sind jedoch widersprüchlich und Problemsituationen diesbezüglich in den vorliegenden Daten selten aufgetreten.

Intelligente Techniken zur Analyse von Lösungsversuchen und Fehlerdiagnose erfordern dagegen die Modellierung einer umfangreichen Wissensbasis für Konzepte, Regeln und typische Fehler (Kap. 3.3 und 3.4). Sind diese ITS mit einem umfassenden Fehlerkatalog ausgestattet, liefern sie für Aufgaben, deren Ziele und Pläne in das Domainwissen eingeflossen sind, häufig beachtliche Diagnoseergebnisse. Da jedoch Variationen und multiple Fehler in einem Programm zu unauflösbaren Mehrdeutigkeiten führen können und Fehlerkataloge stets unvollständig sind, ist der dafür zu investierende Aufwand mittlerweile stark umstritten [vgl. Deek & McHugh 1998, 167]. Analog zu Verfahren, welche ohne modelliertes Expertenwissen die Äquivalenz eines Lösungsversuchs zu einer Referenzlösung bzw. Referenzalgorithmen zu beweisen suchen, kann die Unvollständigkeit zur Einstufung von zulässigen Variationen als Fehler führen [vgl. Deek & McHugh 1998, 167].

Die Beschränkung auf kleine, einfache Aufgabenstellungen oder das 'Erzwingen' eines korrekten Pfades (LISP-Tutor) zur Vermeidung von Mehrdeutigkeiten lassen sich

hingegen kaum mit dem konstruktivistisch-problemorientierten Lernparadigma von VitaminL vereinen. Während Bridge als einziges der vorgestellten ITS explizit auch die Planungsphase der Programmentwicklung unterstützt und gleichzeitig über die sukzessive Auswahl von Zielen und Plänen über ein Menü Mehrdeutigkeiten begrenzt, dürfte die starre Trennung der einzelnen Phasen ein selbstgesteuertes Lernen einschränken. Da hier die Kommunikation nicht systematisch untersucht wurde, liegen nur Hinweise darauf vor, dass Programmplanung bei Anfängern, wie in der Literatur beschrieben (Kap. 2. 2), oft gering ausgeprägt ist. Göldner [2005] zeigt diese mangelnde Planung in der Auswertung der Kommunikation anhand ebenfalls über VitaminL erhobener Daten auf. Wie jedoch die Evaluation von Bridge deutlich macht, reicht das Kennen der Ziele eines Programms nicht aus, um die entsprechenden Sprachkonstrukte zu ermitteln. Gerade bei der aktiven Umsetzung ohne Menü sind dabei die an den vorliegenden Daten beobachteten Schwierigkeiten der 'Übersetzung' in konkreten Code zu erwarten.

Angeregt durch die häufige Nutzung von Beispielen bzw. das Zurückgreifen der Testpersonen auf frühere Lösungen in Problemsituationen soll hier in Ahnlehnung an Neal [1989] (Kap. 3.1) und Weber [1994, 1996] (Kap. 3.4.1) der beispielbasierte Ansatz aufgegriffen und weiterentwickelt werden, um solchen Implementierungsproblemen zu begegnen.

5.2 BEISPIELBASIERTES PROGRAMMIEREN

„Examples are important, if not essential, in learning programming" [Halland & Malan 2004, 83]. Vorgefertigte Beispiele oder eigene frühere Erfahrungen (Fälle) nehmen im Prozess des Lernens sowohl für die Veranschaulichung abstrakter Konzepte als auch für das Übertragen von Problemlöseoperationen auf neue Probleme eine zentrale Rolle ein.

> "In teaching programming, examples can be used for many different purposes. Examples can be used to show the use of a language structure (such as a while loop), to implement an algorithm, as a solution to a problem or to demonstrate programming style …. Examples can also be used to illustrate programming principles, such as object-oriented design. Apart from the direct purpose of illustrating/demonstrating a new concept, an example often has to serve an additional purpose; that of "selling" the concept. There are many concepts in programming that require the student to develop an appreciation of the value of the concept. Take the concept of inheritance for example". [Halland & Malan 2004, 83]

Das Lehren einer Programmiersprache anhand von Beispielen ist daher eine mächtige Strategie und wird insofern auch in Lehrbüchern oft intensiv eingesetzt [Halland & Malan 2004, 83]. Griffiths et al. [1997], Lavy & Or-Bach [2004] und Halland & Malan [2004, 83] betonen dabei die Bedeutung adäquater Beispiele in der Lehre, damit bspw.

objektorientierte Prinzipien und ihre Vorteile tatsächlich verstanden und genutzt werden. Nach Anderson [2000, 246f] werden Problemlöseoperatoren durch Beispiele und analogen Transfer besser erworben als durch Instruktion, da Wissen auf diese Weise aktiv in bestehende Strukturen integriert wird. Die Lösung neuer Probleme durch analoges bzw. fallbasiertes Schließen auf übertragbare Operationen und ihre Anpassung ist für das Problemlösen prinzipiell effizienter als eine zeitaufwendige wissensbasierte Suche und Kombination von einzelnen Operationen [vgl. Reimann & Schult 1996]. Neben Syntax und Semantik einer bestimmten Programmiersprache können Beispiele den Erwerb der häufig hervorgehobenen Pläne in Form von wiederkehrenden Code-Schemata als programmierspezifische Problemlöseoperationen fördern.

> „When these patterns are presented in progressively more sophisticated forms with many examples and student problems at each step, they learn to use these patterns as a model for decomposing a solution into manageable terms (…). These patterns are an example of the problem solving models that help students translate domain specific solutions into computer compatible solutions." [Winslow 1996, 20]

Anfänger stützen sich daher oft auf Beispiele, um Verständnisschwierigkeiten oder Problemsituationen, wie die hier empirisch ermittelten, zu überwinden [vgl. Winslow 1996, 19; Grabowski & Pennington 1990, 48f]. So nannten die Studierenden in der Umfrage von Ala-Mutka et al. [2005] Beispielprogramme als hilfreichste Ressource während der Aufgabenbearbeitung. Neal [1989] berichtet von sehr positivem Feedback für ihre beispielbasierte Programmierumgebung, in der Lernende Beispiele aus einer Beispielbibliothek besonders bei Unsicherheiten bezüglich Syntax und Semantik, weniger auch zur Hilfe beim algorithmischen Design aufgerufen haben. Für Anfänger zeigt sich jedoch gerade das Abrufen bzw. Erkennen geeigneter Beispiele als Fehlerquelle. So verfügen Anfänger einerseits per Definition über wenig Erfahrung im Sinne von Programmierepisoden. Andererseits orientieren sie sich häufig zu stark an oberflächlichen Merkmalen (siehe Kap. 2.3.2, [vgl. auch Anderson 2000, 250]), so dass hilfreiche Beispiele nicht zugänglich sind oder nicht genutzt werden [vgl. Weber 1994]. Auch in den hier analysierten Daten wurden nicht immer angemessene Beispiele herangezogen, so dass eine höhere Abstraktions- und Adaptionsleistung erforderlich gewesen wäre, um diese Beispiele als Hilfestellung einzusetzen.

An dem Punkt der Erkennung ähnlicher Beispiele setzt das nun zu beschreibende Verfahren an. Während Programmieranfänger bei Neal [1989] Beispiele aktiv selbst auswählen und ELM-PE [Weber 1994, 1996] frühere Programmierepisoden im Wesentlichen zur Einschränkung des Suchraums bei der Erklärung eines Fehlers anwendet,

wird hier das automatische Bereitstellen ähnlicher Beispiele in Problemsituationen als Ziel gesetzt. Im Gegensatz zu ELM-ART [Brusilovsky et al. 1996], dem netzbasierten Nachfolger von ELM-PE, in dem Beispiele ebenfalls als Hilfestellung geboten werden, soll jedoch kein explizites Domainwissen modelliert werden. Beispiele und Problemstellungen sind in ELM-ART anhand eines 'LISP conceptual network', einem semantischen Netz aus 'is-a'- oder 'part-of'-Relationen zwischen Lispkonzepten, indexiert. Stattdessen soll mithilfe maschineller Lernverfahren die Bereitstellung geeigneter Beispiele aufgrund gleicher, 'einfacher' Code-Merkmale realisiert werden[62].

Nach dem Vorschlag von Neal [1989] werden syntaktische und semantische Merkmale genutzt, um Beispielprogramme für Sprachkonstrukte und Algorithmen geben zu können. Anhand des Quelltextes der aktuellen Implementierung wird ein ähnliches, korrektes Programm aus einer Fallbasis abgerufen und den virtuellen Teams als Dokument zur Verfügung gestellt. Diese Datenbasis umfasst momentan ausführlich kommentierte Beispielprogramme mit einem Umfang von 'HelloWorld.java' bis zu Programmen, die eine Datei einlesen und anschließend Wörter aus dem Text löschen. Je nach Fortschritt in der Kodierung können somit komplexere Beispiele präsentiert werden. Mithilfe dieser Beispiele sind nicht nur Hinweise auf richtige Syntax und Semantik bei der Umsetzung der aktuellen Pläne, sondern auch Anregungen für die Komposition einzelner Anweisungen übertragbar.

Im Hinblick auf den Transfer aus den Beispielen könnte sich dabei gerade die Gruppensituation als günstig erweisen, da der für das erfolgreiche Lernen aus Beispielen entscheidende Selbsterklärungseffekt[63] [vgl. Bassok et al. 1989] durch die Kommunikation über die Beispiele mit den anderen Teammitgliedern gefördert werden dürfte. Zwar können Fehler durch Beispiele nicht automatisch diagnostiziert werden, jedoch entfällt der immense Aufwand für die Entwicklung einer ausreichend vollständigen Wissensbasis. Das System wird besser, indem 'lediglich' neue Beispielprogramme der Datenbasis hinzugefügt werden. Zudem kann anders als bei anderen Verfahren auch zu syntaktisch fehlerhaften Lösungsversuchen Unterstützung angeboten werden. Angesichts der freien Kodierung sollten Syntaxfehler insofern toleriert werden können, dass Hilfe-

---

[62] MEDD hingegen setzt maschinelle Lernverfahren zur Erstellung eines Fehlerkatalogs ein. Nach einer intensionsbasierten Diagnose ähnlich der Diagnose in PROUST (Kap. 3.3.1) werden dabei nicht zu erklärende Abweichen im Code konzeptionell zu Fehlerklassen geclustert [vgl. Cruz & Sison 2002].

[63] Bassok et al. [1989] fanden in ihrer Studie, dass Lernen aus Beispielen erfolgreicher ist, wenn sich der Lernende die Beispiele selbst erklärt. Jones & VanLehn [1993] führen diese Ergebnisse darauf zurück, dass durch die Elaboration von Beispielen Wissenslücken aufgedeckt und das entsprechende Wissen erworben wird.

stellung bspw. zur Nutzung einer Methode nicht erst nach dem Entfernen aller Syntaxfehler möglich ist.

5.3 EVALUATION DES ANSATZES

Zur Evaluation des beispielbasierten Ansatzes ist zunächst die Nützlichkeit von Beispielen als Unterstützung in Problemsituationen zu überprüfen. Dazu bewerteten Teams in Benutzertests, wie hilfreich die ihnen bei Problemen – manuell – bereitgestellten Beispiele im Lösungsprozess waren.

5.3.1 Beschreibung WOz-Test

Da eine empirische Evaluation zur Unterstützung des Programmierprozesses von Anfängern durch Beispiele nur in der Studie von Neal [1989] vorliegt, wurden Benutzertests mit virtuellen Teams im Rahmen des VitaminL-Projekts durchgeführt. Dabei stand weniger die konkrete Implementierung als vielmehr die Überprüfung des prototypischen Systementwurfs und des beispielbasierten Ansatzes im Vordergrund. Daher wurde eine Simulation einer tutoriellen Komponente nach Art eines 'Wizard-of-Oz'-Experiments bzw. einer 'Hidden-Operator-Simulation' als Untersuchungsdesign gewählt, d.h. die Funktionsweise eines hypothetischen Systems wird durch einen menschlichen Operator simuliert. Die Benutzer gingen davon aus, mit einem maschinellen System zu interagieren. Somit ist es möglich, ein nicht vollständig realisiertes System unter relativ realistischen Bedingungen zu testen [vgl. Womser-Hacker 2004]. Zudem können zusätzliche Anforderungen ermittelt werden, die in dem bisherigen Entwurf nicht ausreichend berücksichtigt sind.

Fünf Teams bearbeiteten innerhalb der VitaminL-Plattform ca. 90 Minuten lang eine Programmieraufgabe, wobei ihnen der Tutor in Problemsituationen Beispiele als Hilfestellung präsentierte. Zwei der Teams bestanden aus Hildesheimer Studenten, die an der virtuellen Kooperationsveranstaltung teilgenommen und für die Bearbeitung einer Semesteraufgabe sich als Gruppe neu formiert haben. Ein Team arbeitete weiterhin rein virtuell zwischen Konstanz und Hildesheim, zwei weitere Teams waren Gruppen der traditionellen Java-Vorlesung mit Begleitübung.

Die Tests fanden nach Ende der Vorlesungszeit (nach etwa fünf Monaten Java-Erfahrung) zeitgleich zu einer selbstständigen Projektentwicklung statt. Je nach Fortschritt in der Projektbearbeitung war von einem unterschiedlichen Kenntnisstand der

Teilnehmer auszugehen, so dass sie nur mit Vorbehalt als Anfänger zu bezeichnen sind. Um dennoch alle Teams vor realistische Problemsituationen stellen zu können, die mithilfe der Beispiele zu überwinden waren, wurde versucht, eine relativ schwierige und unbekannte Aufgabe auszuwählen. Die ermittelten Fehler und Probleme mit Arrays, Schleifenbedingungen und Rückgabewerten dienten hierbei als Orientierung. Aus einem Text sollten Stoppwörter wie Konjunktionen, Präpositionen oder Artikel gelöscht und der Text ohne diese Stoppwörter ausgegeben werden. Da die Hildesheimer Studenten aus der Lehrveranstaltung 'Einführung in die Informationswissenschaften' den Praxisbezug der Stoppworttilgung im Information Retrieval kannten, waren Schwierigkeiten mit dem Verständnis der Aufgabenstellung nicht anzunehmen. Diese Einschätzung bestätigte sich in den Benutzertests.

Den Teams wurde in einer kurzen Einführung das Ziel des Benutzertests, die Evaluierung eines ersten Prototyps für eine automatische tutorielle Komponente, erklärt. Sie könnten diesen elektronischen Tutor, der als weiteres Teammitglied eingeloggt war, jederzeit mithilfe einer Chatnachricht zur Unterstützung auffordern. Während der Tutor auch von sich aus auf vermutete Problemsituationen reagieren würde, stände der virtuell anwesende Versuchsleiter nur für Fragen in Bezug auf den Ablauf des Tests und die Benutzung der Plattform zur Verfügung. Zudem wurden die Teilnehmer gebeten, für die spätere Evaluation auf die Fragen des Tutors zur Nützlichkeit einer jeden geleisteten Hilfestellung ein kurzes – intellektuell auswertbares – Feedback zu geben. Für den eigentlichen Test wurden die Teilnehmer daraufhin auf verschiedene Räume verteilt, so dass auch der Versuchsleiter unbeobachtet den Tutor simulieren konnte[64].

Trat während der Aufgabenbearbeitung eine mögliche Problemsituation auf – als Hinweise dienten die Indikatoren Zeit, Kommunikation und Fehler im Quell-Code[65] – öffnete der elektronische Tutor ein vorbereites Beispieldokument in der Plattform. Dabei wurde entsprechend des aktuellen Systementwurfs versucht, anhand des Quell-Codes der Teammitglieder aus der vorbereiteten Datenbasis ein Beispiel auszuwählen, welches die gleichen Attribute – bspw. Methodennamen oder Datentypen – aufwies. Als kritisch erwies sich hierbei nicht nur die „getreue Abbildung des Systems“ [Womser-

[64] Der Benutzertest mit der virtuellen Gruppe aus Hildesheimer und Konstanzer Teammitglieder wurde vollständig virtuell durchgeführt. Dazu wurden die Instruktionen vorab per Email an die Teilnehmer versandt und Fragen diesbezüglich zu Beginn der Sitzung besprochen. Der Bewertungsbogen wurde nach dem Test ebenfalls als Email verschickt.

[65] Die Indikatoren wurden nicht durch konkrete Werte formalisiert, da in erster Linie geprüft werden sollte, ob Beispiele als Hilfestellung nützlich sind und weniger, wann der Tutor eingreifen sollte.

Hacker 2004] durch den menschlichen Tutor, sondern auch die Anzahl der verfügbaren Beispiele.

Die Beispielkollektion umfasste ca. 70 kommentierte Java-Dateien, von denen sich 30 direkt auf Funktionen bezogen, die zur Lösung der Aufgabenstellung anwendbar waren[66]. Einfache Beispiele zeigten die Nutzung von 'indexOf' zur Ermittlung eines Wortes aus einem String oder das Zählen von Wörtern in einem String mithilfe eines 'StringTokenizers'. Komplexere Beispiele zeigten hingegen einen Großteil einer möglichen Lösung. Aufgrund der geringen Beispielzahl und nicht antizipierter Lösungsversuche konnten nicht immer geeignete Beispiele angeboten werden. Die Reaktionszeit des simulierten Tutors wurde durch diese Problematik stark verzögert[67].

Die Kommunikation des simulierten Tutors beschränkte sich im Wesentlichen auf die Nachfrage, ob ein Problem vorläge und das Team Hilfe in Form eines Beispiels erhalten wolle, sowie Hinweise darauf, dass ein Beispiel als Dokument geöffnet wurde. Nach Darbietung eines jeden Beispiels fragte der Tutor zudem, wie hilfreich das Beispiel war und ob ggf. ein weiteres benötigt wurde. Während die klassische Anwendung derartiger Simulationstests in der Entwicklung natürlichsprachiger Dialogsysteme auf die Kommunikation zwischen Mensch und Maschine abhebt ('Computer-Talk') [vgl. Womser-Hacker 2004], wurde hier primär die Verwendung und Nützlichkeit der bereitgestellten Beispiele ausgewertet. Qualitative Beobachtungen zur Interaktion mit dem Tutor während der Sitzung wurden jedoch angemerkt.

5.3.2 Ergebnisse Nützlichkeit

Die Evaluation beruht sowohl auf den während der Sitzung gegebenen Bewertungen zu den einzelnen Beispielen als auch auf einem nach dem Test auszufüllenden Fragebogen. In diesem Fragebogen sollten auf einer vierstufigen Skala ('sehr' bis 'gar nicht' bzw. 'immer' bis 'nie') die Effektivität, Verständlichkeit und Angemessenheit der Beispiele sowie der Lerneffekt und das Maß an Unterstützung beurteilt werden. Zusätzlich wurde nach weiteren nützlichen Hilfestellungen, der idealen Interaktion mit einem elektronischen Tutor, eigenen Problemlösestrategien sowie sonstiger Kritik und Anregungen gefragt.

---

[66] Grundsätzlich sind zur Erfüllung der Aufgabenstellung zwei Lösungswege zu unterscheiden. Eine Möglichkeit ist das Zerlegen des Textes über einen 'StringTokenizer' mit anschließendem Vergleich zu dem Array mit den Stoppwörtern. Auch das direkte Löschen aus dem Text mithilfe der Methoden eines 'StringBuffers' kann eingesetzt werden.

[67] Dieser Aspekt führte jedoch zu keiner negativen Kritik der Testpersonen.

5.3.2.1 BENUTZERTESTS

In Anlehnung an die Kategorisierung von Anfängerproblemen soll qualitativ differenziert werden, inwieweit sich Beispiele für die Überwindung von syntaktischen, semantischen, logischen oder die Komposition betreffende Problemsituationen eignen. Dabei sind semantische Probleme im hier definierten Sinne nicht aufgetreten, vor allem da nur zwei der Gruppen im Laufe des Tests ausführbaren Code implementierten. Insgesamt wurden den fünf virtuellen Teams 19 Beispiele präsentiert, etwa drei - vier pro 90minütiger Sitzung. In mehreren Situationen, ähnlich der Abbildung 18, scheint das Beispiel als Referenz für die korrekte Syntax einer Anweisung gedient zu haben. Hinweise auf derartigen Transfer gibt die Navigation der Teammitglieder zwischen den verschiedenen Dokumenten in der Plattform, welche anhand der Logfiles nachvollzogen werden kann.

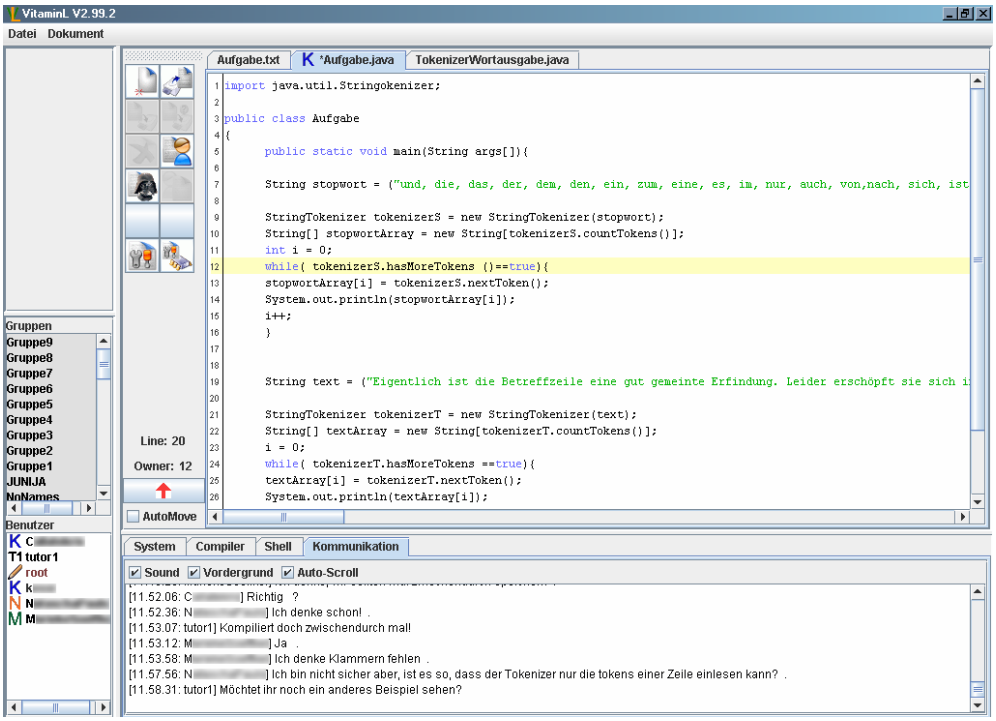

Abbildung 18: Screenshot WOz-Test

In einem Fall scheint ein Teammitglied sukzessive die Fehler in seinem Programm zu beheben, indem er nach Fehlermeldungen im Wechsel zwischen den Beispielen und seinem Dokument Syntaxfehler – wie Klammersetzung, fehlende Parameter der 'Main'-Methode, fehlendes Semikolon – entfernt. Im Gegensatz dazu war das Korrigieren syntaktischer Fehler, besonders im Hinblick auf Klammersetzung und 'String'- bzw. 'StringArray'-Initialisierung, nach der Darbietung der Beispiele hingegen zumeist als 'nebenläufig' zu beobachten. Oftmals wurden Syntaxfehler im Code scheinbar als un-

problematisch angesehen, da die Suche nach einem möglichen Lösungsansatz für die Aufgabenstellung oder spezielle Teilziele wie das Löschen aus einem String im Vordergrund stand. Diese Einschätzung spiegelt sich an dem fortschreitenden, syntaktisch fehlerhaften Kodieren ohne zwischenzeitliches Kompilieren, aber auch in der seltenen systematischen Korrektur mithilfe der Beispiele. Insofern nicht eine Problemsituation zu einem konkreten Syntaxfehler vorlag, wurden 'rein' syntaktisch bzw. semantisch ähnliche Beispiele zumeist als nicht hilfreich oder lediglich ein bisschen hilfreich bewertet. Einen solchen Fall zeigt die Abbildung 19 in der alle drei Teilnehmer anhand des einfachen Beispiels 'StringsVerbinden.java'[68] die vergessene 'Main'-Methode und die Notation für die 'String'-Initialisierung in ihren Dokumenten korrigierten. In diesem Fall verwirrte die Nutzung der in dem Beispielprogramm enthaltenen Methode 'concat' jedoch zusätzlich. Ursachen für die Unsicherheiten können sowohl die Unkenntnis der Methode sein als auch die Annahme, dass ein Beispiel immer den einzuschlagenden Lösungsweg benennt.

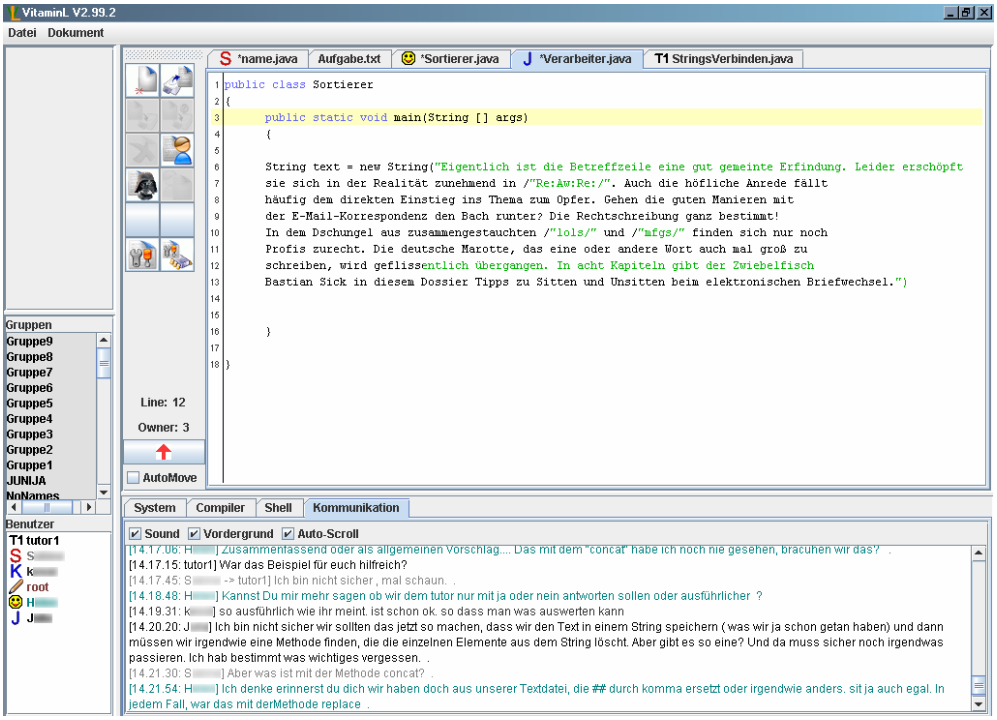

Abbildung 19. Screenshot WOz-Test

Syntaktische Problemsituationen, in denen die Beispiele keine Hilfestellung boten, waren ebenfalls festzustellen. Während für die Illustration der String-Notation mit eingeschlossenen Anführungszeichen über den Backslash-Operator kein Beispiel zur

[68] In diesem Beispiel werden zwei Strings erzeugt und aneinander angefügt sowie ausgegeben.

Verfügung stand, konnte die Fehlerursache aus dem entsprechenden Beispiel nicht abgeleitet werden (Abbildung 20). Der Tutor wurde zu diesem Problem befragt, konnte jedoch kein geeignetes Beispiel liefern, da das aktuell bereitgestellte den betreffenden Code bereits enthielt.

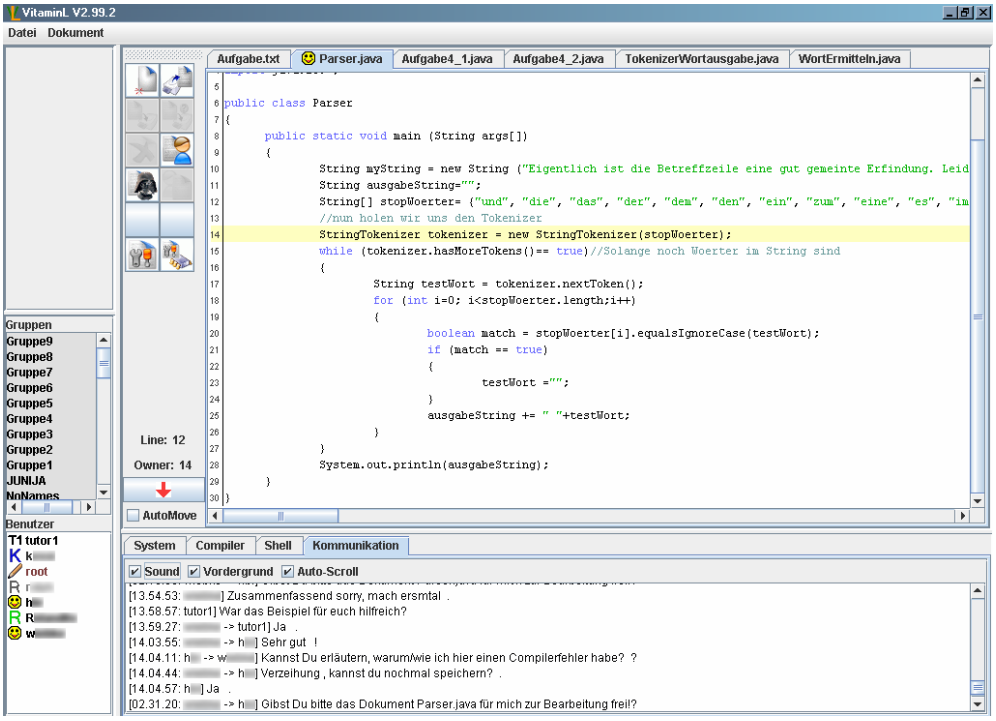

Abbildung 20. Screenshot WOz-Test

Nach 15 Minuten Unschlüssigkeit führte letztlich der Tipp des Experimentleiters *„Was wollt ihr denn 'tokenizen'?“* zu der sofortigen Identifikation des Fehlers.

Die einzige Problemsituation die Logik des Programms betreffend konnte mithilfe eines geeigneten Beispiels überwunden werden (Abbildung 21). Durch eine fehlerhafte Komposition entstand eine, an der Kommunikation zu erkennende, Stocksituation. Erst durch die Übernahme und Anpassung des offen gelegten Beispiel-Codes war die Problemsituation zu überwinden.

Da zu diesem Zeitpunkt der Test aus zeitlichen Gründen abgebrochen wurde, kann jedoch nicht rekonstruiert werden, ob die Fehlerursache auch verstanden wurde. Die positiv bewerteten Beispiele, etwa die Hälfte der gezeigten Beispiele[69], waren Programme, welche die weitere Planung z. B. bezüglich der Umsetzung eines Tokenizers oder des Einsatzes einer Schleife beinhalteten (Abbildung 22).

[69] Von den 19 Beispielen wurden zehn als hilfreich ('ja' oder 'sehr gut'), zwei als 'ein wenig' hilfreich und vier als 'ungeeignet' bewertet. Zu drei Beispielen erfolgte, u. U. nach wiederholter Nachfrage des Tutors, kein explizites Feedback.

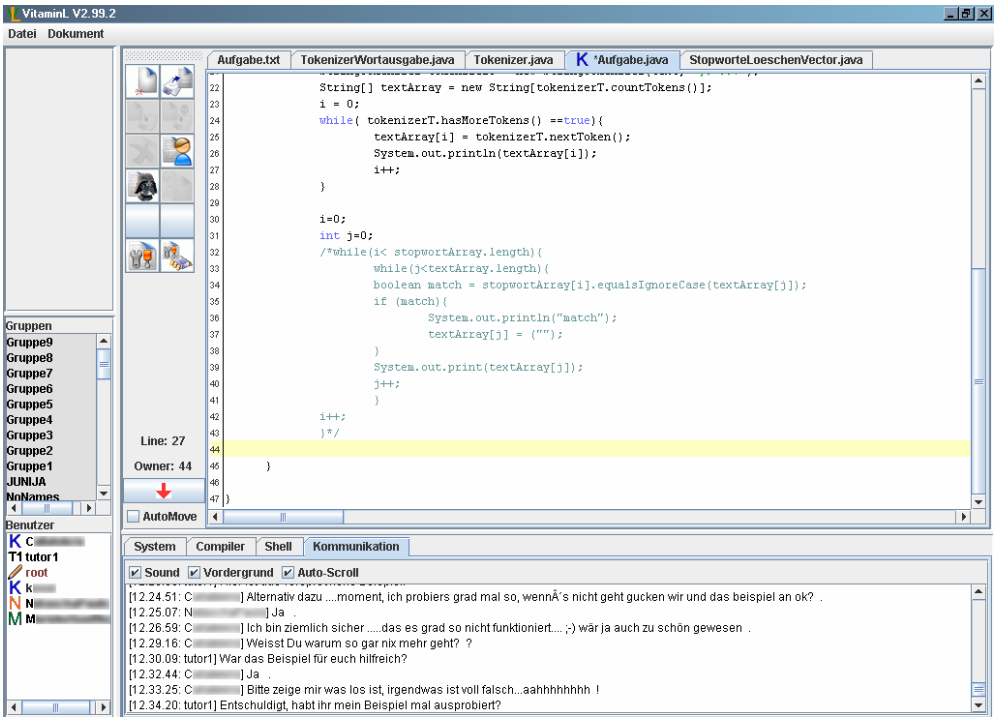

Abbildung 21. Screenshot WOz-Test

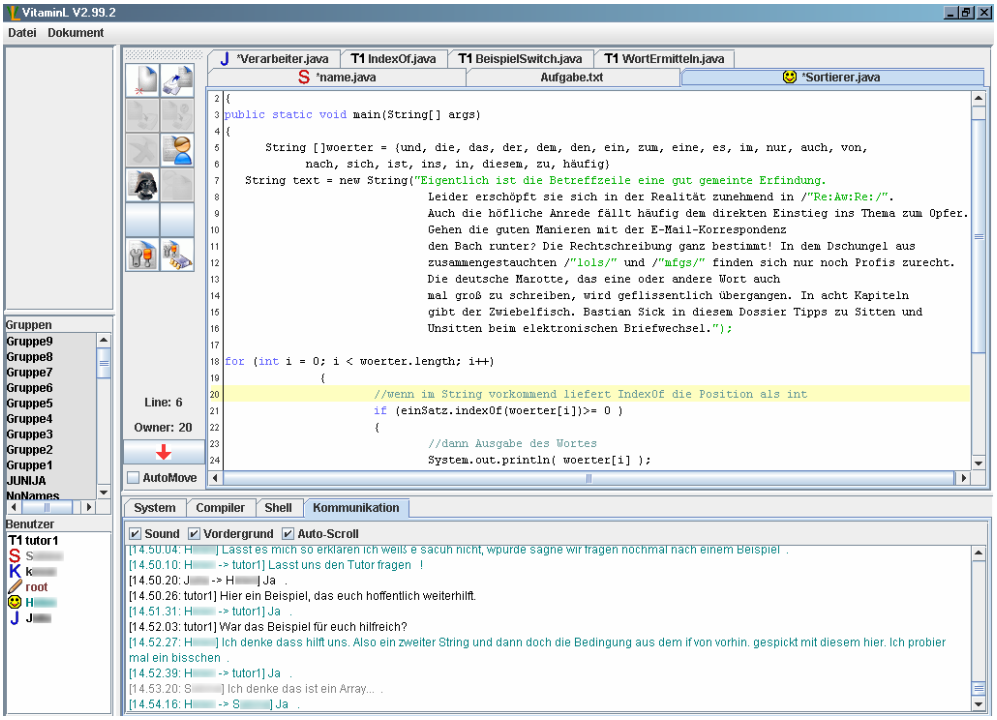

Abbildung 22. Screenshot WOz-Test

Diese Problemsituationen bezüglich des algorithmischen Designs und anwendbarer Methoden lassen sich jedoch nicht immer am Quelltext ablesen. Während in einem Team offensichtlich in Folge mangelnder Planung bzw. Kommunikation Methodennamen 'erstmal' hingeschrieben wurden bevor ihre Brauchbarkeit bzw. die korrekte Einbindung

zur Frage stand, waren Hinweise oftmals nur der Kommunikation zu entnehmen. So wurde früh deutlich, dass eine alleinige Orientierung des Tutors am Code für eine effektive Unterstützung nicht ausreichend ist, so dass auch auf Schlüsselwörter in der Kommunikation reagiert wurde. Die positive Resonanz auf solche 'vorgreifenden' Beispiele scheint die Wichtigkeit des Identifizierens solcher Problemsituationen, in denen lange Zeit nicht kodiert wird, zu bestätigen. Obwohl in einer Sitzung ohne Tutor auf diese Stocksituationen möglicherweise nach gewisser Zeit ein Kodierungsversuch – möglicherweise nach 'Versuch-und-Irrtum' – folgen würde.

Diese Versuche könnten wiederum einer Voraussage eines passenden Beispiels dienen. Ferner ist der Aspekt der Komplexität der bereitgestellten Beispiele zu diskutieren, d.h. ob unter didaktischen Gesichtspunkten auch umfangreiche Beispiele bereitgestellt werden sollten oder lediglich Beispiele, die nur einen nötigen Schritt im Problemlöseprozess zeigen. So stellt sich weiter die Frage, ob ein tutorielles System auch auf Ansätze der Testpersonen reagieren sollte, die den Problemlöseprozess nicht korrekt verfolgen. In einem Fall scheint das Darbieten eines entsprechenden Beispiels auf Kommunikation über die Verwendung eines 'switch-case' teilweise als Bestätigung des richtigen Ansatzes interpretiert worden zu sein (Abbildung 23).

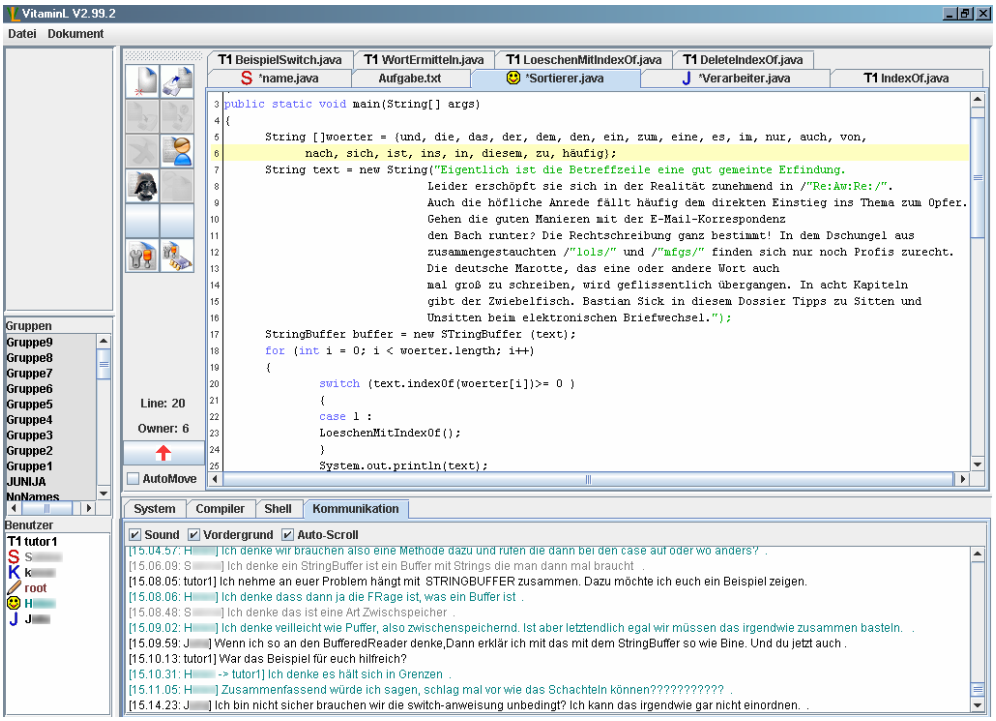

Abbildung 23. Screenshot WOz-Test

Generell ist festzustellen, dass über die Beispiele nur wenig oder überhaupt nicht kommuniziert wurde. Die Beobachtungen aus den fünf Benutzertests zeigen, dass Beispiele

großteils geeignet sind, um syntaktische Fehler aus ihnen abzuleiten und die Komposition von Anweisungen zur Erreichung von Teilzielen einer Aufgabe zu illustrieren. Um beides realisieren zu können, sollte ein beispielbasiertes System idealerweise sowohl auf syntaktischen bzw. semantischen Ähnlichkeiten zum Quell-Code beruhen als auch Beispiele auf Stichworte hin bereitstellen. Für spezifische Fehler, die nicht durch Transfer identifiziert werden können, wären jedoch bspw. Techniken des automatischen Debuggens nötig.

5.3.2.2 FRAGEBÖGEN

Auch die nach dem Test ausgefüllten 13 Fragebögen[70] spiegeln eine durchaus positive Reaktion auf die beispielbasierte Unterstützung.

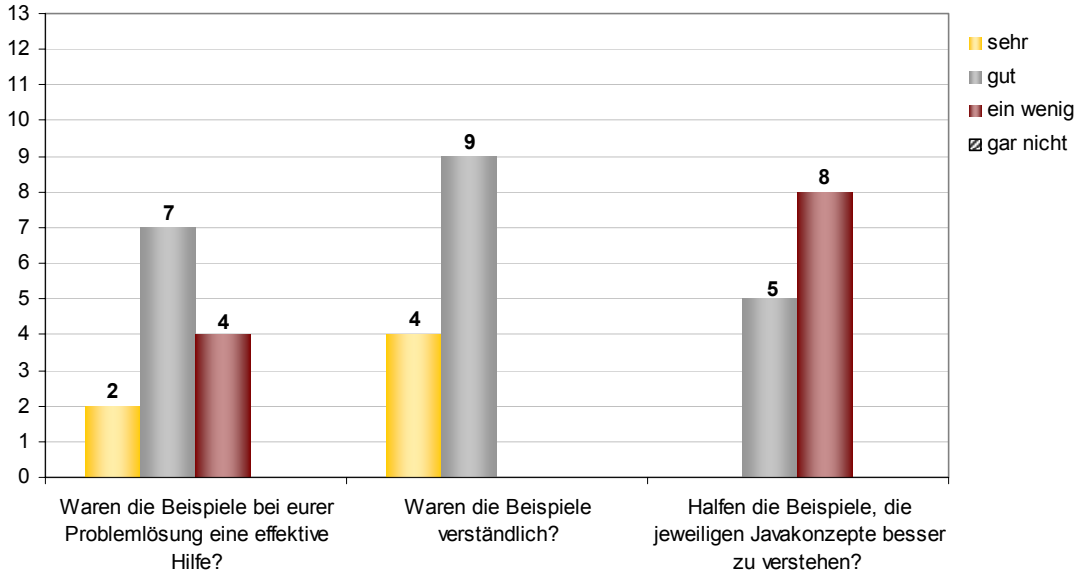

Abbildung 24. Ergebnisse zur Einschätzung von Effektivität, Verständlichkeit und Lerneffekt

Wie Abbildung 24 zeigt, bewerteten zwei der 13 Teilnehmer die Beispiele im Hinblick auf die Effektivität dieser Hilfestellung für den Problemlöseprozess als 'sehr gut', sieben Teilnehmer als 'gut' und vier Teilnehmer sahen die Beispiele als 'ein wenig' hilfreich. Keiner der Teilnehmer empfand die Beispiele als 'gar nicht' hilfreich oder 'gar nicht' geeignet, die jeweiligen Java-Konzepte für ein besseres Verständnis zu veranschaulichen. Daraus ergibt sich für die Effektivität dieser Unterstützungsform ein Mittelwert von 2, 15 ('gut') mit einer Standardabweichung von 0, 69 und für den Lerneffekt durch

[70] Von den 15 Fragebögen konnten nur 13 ausgewertet werden, da ein Teilnehmer in der Bewertung der Effektivität der Beispiele sowohl 'sehr' als auch 'gut' angegeben und ein anderer Teilnehmer keines der Beispieldokumente in der Plattform wahrgenommen hatte. Letzteres war aus dem Kommentar unter der Frage 'Sonstige Anregungen bzw. Kritik' ersichtlich.

die Beispiele ein Mittelwert von 2, 62 ('gut' - 'ein wenig') mit Standardabweichung 0, 51. Aufgrund der geringen Teilnehmerzahl scheint die Auswertung weiterer statistischer Maße nicht sinnvoll. Trotzdem kann gefolgert werden, dass Beispiele eine sinnvolle Unterstützungsform für die Überwindung von Problemen darstellen.

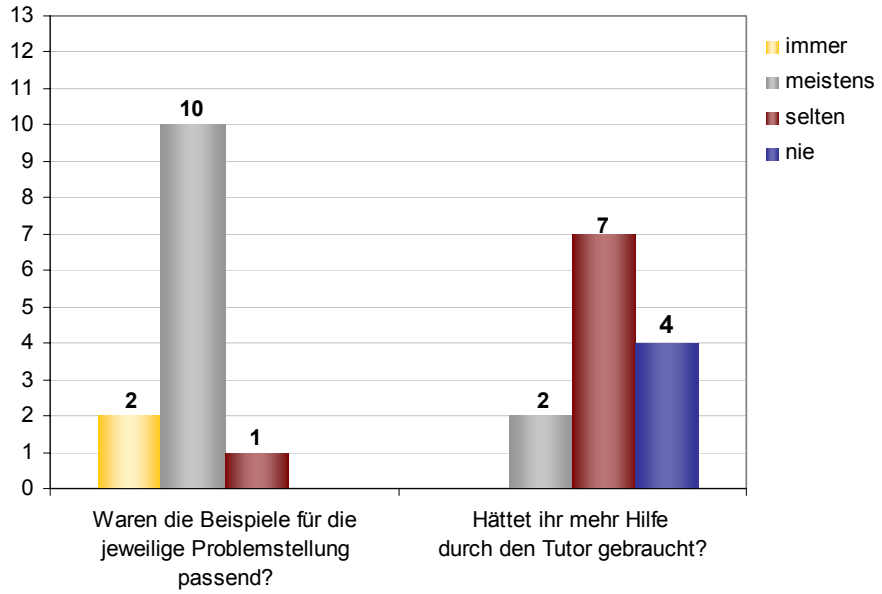

Abbildung 25. Ergebnisse zu Relevanz und Ausmaß der Hilfe

Damit Beispiele eine effektive Unterstützung darstellen können, müssen sie der jeweiligen Problem- bzw. Fragestellung entsprechen und als geeignet erkannt werden. Aus Abbildung 25 wird deutlich, dass ein Großteil der Teilnehmer (zehn von 13) die präsentierten Beispiele 'meistens' (Mittelwert 1, 92; Standardabweichung 0, 49) als passend angesehen hat. Dies lässt zwar für die erste prototypische Implementierung noch keine Aussagen zu, da sich die intellektuelle Auswahl nicht ausschließlich am Quelltext orientierte. Dennoch scheint die Orientierung am aktuellen Code der Programmierer prinzipiell ein nützlicher Ansatz. Hier zeigt sich jedoch eine Abweichung zur Einschätzung der Effektivität. Obwohl in der globalen Bewertung der Beispiele zehn Teilnehmer sie als 'meistens' passend einstuften, waren sie nur für sieben Teilnehmer auch eine 'gute' Hilfestellung.

Dem gemäß nannten drei Personen in der offenen Frage nach anderen sinnvollen Hilfestellungen den Wunsch, *„jemand ansprechen (zu) können, der einem speziell weiterhilft“* oder *„ein Tutor, der flexibler reagiert und man konkrete Fragen stellen kann“*. *„Mir würde hilfreich erscheinen, wenn man noch Tipps von einem echten Tutoren bekommt. Die Beispiele sind sehr hilfreich, manchmal ist eine direkte Antwort jedoch*

hilfreicher“. Beispiele können demnach Anregungen und Implementierungshilfe geben, sind für bestimmte spezifische Problemsituationen jedoch allein nicht ausreichend.

Vier Teilnehmer nannten als andere Hilfestellung die API, wobei nur in einem Falle konkreter darauf eingegangen wurde, in welcher Form dies realisiert werden könnte: *„ein(e) in das Programm eingebundene(s) API(Nachschlagewerk)“*. Zwei andere Vorschläge betrafen speziell den Editor: automatisches Setzen der zweiten Klammer, automatisches Einrücken, das Markieren gleicher Elemente auf Anklicken hin sowie *„evtl. WYSIWYG Editor“*. 'Reale' Kommunikation über Internet-Telefonie und die Anbindung an ein Java-Forum wurden ebenfalls als sinnvolle Erweiterungsmöglichkeiten betrachtet. Drei der Teilnehmer gaben an, dass es „*schon gut so“* war oder ihnen *„grade wirklich keine anderen ein(fallen)“*.

In Bezug auf das ideale Eingreifen eines Tutors wurden sowohl inhaltliche Aussagen darüber, dass die Beispiele *„schon relativ passend und hilfreich“* waren, als auch über die Interaktionsform getroffen. Das Nachfragen des Tutors, wenn das Team lange nichts geschrieben hatte bzw. „*nicht weiter kommt“*, empfanden zwei Teilnehmer als *„gar nicht schlecht“* bzw. *„schon gut und ausreichend“*. Drei Teilnehmer könnten sich die tutorielle Unterstützung auf *„konkrete Suchworte“*, Schlüssel- bzw. Stichwörter hin vorstellen. Ein Teilnehmer schlug vor, dass der Tutor, wenn *„etwas nach einigen Zeilen nicht zum Richtigen führen würde“* in der Klasse der Gruppe Verbesserungsvorschläge unterbreitet statt immer neue Beispiele in neuen Klassen zu zeigen. Eine andere Person hingegen fände *„verschiedene Angebote von Quellcodes“* sehr sinnvoll. Sechs der 13 Teilnehmer würden es vorziehen nicht von einem Tutor 'gestört' zu werden, sondern ihn bei Bedarf anzufordern.

Die hohe Motivation, Probleme möglichst selbst zu lösen, zeigt sich auch in Abbildung 25. Elf von 13 Teilnehmern hätten 'selten' oder 'nie' mehr Hilfe durch den Tutor benötigt. Sowohl die positive Aufnahme der Beispiele als Ressource für eigenständigen Transfer als auch die Nennungen zu weiteren nützlichen Hilfestellungen, zusätzlich zum Tutor, fügen sich in dieses Bild ein. Dabei spielt das Arbeiten im Team scheinbar eine zentrale Rolle. Auf die Frage, wie sich die Teams gewöhnlich bei Problemen weiterhelfen, betonten vier Teilnehmer explizit die gemeinsame Lösungsfindung in der Gruppe. Das Nachschlagen in Büchern (acht Nennungen) und in der Klassenbibliothek (sieben Nennungen) waren die Strategien, die am häufigsten genannt wurden. Das Suchen und Ausprobieren von Beispielen (vier Nennungen) oder alten Übungsaufgaben

(zwei Nennungen) sowie das Internet allgemein (fünf Nennungen) wurden des Weiteren als Wege zur Überwindung von Problemsituationen angeführt. Fragen an den menschlichen Tutor wurde hingegen nur zweimal angegeben. Zumindest zu dem untersuchten Kenntnisstand der Teilnehmer deutet dies auf erfolgreiches selbstgesteuertes Lernen durch aktive Informationsbeschaffung und Wissenstransfer im Team[71], so dass auch die Bereitschaft für die eigene Transferleistung aus Beispielen offensichtlich großteils gegeben ist. Antworten auf die Frage nach sonstigen Anregungen und Kritik betrafen fast ausschließlich die Form der strukturierten Kommunikation, die für die hier untersuchte Fragestellung jedoch nicht von Bedeutung ist.

5.4 IMPLEMENTIERUNG EINES PROTOTYPS ZUR AUTOMATISCHEN BEISPIELLIEFERUNG

Die Auswertung der Benutzertests und Fragebögen verdeutlicht das positive Feedback der Testpersonen auf den Ansatz des beispielbasierten Programmierens. Die angebotenen Beispiele wurden zumeist als sinnvoll bewertet und auch ihr Code teilweise übernommen. Hierauf aufbauend soll die prototypische Implementierung eines Systems skizziert werden, das anhand syntaktisch-semantischer Merkmale des Quelltextes der Programmierer automatisch geeignete Beispiele bereitstellt. Dazu werden maschinelle Lernverfahren in **Weka**[72] (Waikato Environment for Knowledge Analysis) auf ihre Eignung für die Identifikation hilfreicher Beispiele experimentell überprüft. Aufgrund zahlreicher Varianten der Implementierungsmöglichkeiten und multipler Syntaxfehler scheinen ähnlichkeitsbasierte Ansätze sinnvoller als ein exaktes Abgleichen bspw. zu einer Referenzlösung (vgl. Kap. 3).

5.4.1 Maschinelles Lernen mit Weka

Ziel des maschinellen Lernens ist es, ein System zu befähigen, durch Erfahrung die gleichen Aufgaben zunehmend besser und effizienter zu lösen [vgl. Simon 1983, 28]. Bei der Realisierung solcher Systeme, „die in der Lage sind, durch die Benutzung von Eingabeinformation neues Wissen zu konstruieren oder bereits vorhandenes Wissen zu verbessern“[Herrmann 1997, 15], kommen unterschiedliche Lernstrategien zum Einsatz. Neben Auswendiglernen und direkter Eingabe, kann Lernen durch Analogie, Deduktion oder Induktion erfolgen [vgl. Herrmann 1997, 16f].

---

[71] Dies konnte besonders bei den Teilnehmern festgestellt werden, die bereits die virtuelle Kooperationsveranstaltung besucht hatten.

[72] Weka ist frei erhältlich unter der Projektseite: http://www.cs.waikato.ac.nz/ml/weka/

Beim Analogielernen wird neues Wissen erworben, indem es über ähnliche frühere Erfahrungen an eine neue Situation angepasst und dieses transformierte Wissen die Wissensbasis erweitert. Diese Strategie nutzt bspw. das 'Case-Based-Reasoning' (fallbasiertes Schließen). Zu einer neuen Situation wird der ähnlichste Fall oder mehrere ähnliche Fälle aus der Fallbasis abgerufen und die Lösung(en) mithilfe von generellem Domänenwissen so adaptiert, dass sie den Eigenschaften des aktuellen Problems entsprechen [vgl. Althoff 2001].

Deduktives Lernen bezeichnet das Anwenden von vorhandenem (Regel-)Wissen und Schlussregelmechanismen, um konkrete Eingabewerte oder Beispiele der Wissensbasis zu analysieren und dadurch bestehendes Wissen zu reorganisieren bzw. neues Wissen zu akquirieren [vgl. Herrmann 1997, 16]. Eine Form deduktiven Lernens ist das in ELM (Kap. 3.4.1) umgesetzte erklärungsbasierte Lernen, bei dem Regelwissen (über die Programmierdomäne) genutzt wird, um zu erklären, warum ein Beispiel einem Zielkonzept (bspw. 'ELSE-Test') angehört. Die Erklärung ermittelt dabei die für die Zugehörigkeit relevanten Eigenschaften eines Beispiels, so dass Beispiele nach logischen Prinzipien generalisiert werden können [Mitchell 1997, 307].

Im Gegensatz dazu ist beim induktiven Lernen kein explizites Wissen a priori vorhanden, sondern es wird durch Extraktion impliziter Strukturen aus Beispielen neues Wissen gewonnen. Induktives Lernen kommt bspw. im 'Data Mining' zum Einsatz, dem „Erwerb struktureller Beschreibungen aus Beispielen“ [Frank & Witten 2001, XIV]. Hierbei sollen Strukturen und Beziehungen von Eigenschaften in großen Datenmengen identifiziert werden, um durch Generalisierung dieser Muster Vorraussagen für neue Daten zu treffen [vgl. Mitchell 1997, 307; Frank & Witten 2001, 7f].

Beim induktiven Lernen lassen sich unüberwachte Verfahren und überwachte Verfahren differenzieren. Bei Unüberwachten Lernverfahren werden durch Entdeckung gemeinsamer Merkmale mittels 'Clustering' Gruppen zusammengehöriger Beispiele gebildet [vgl. Frank & Witten 2001, 42f]. Lernt das System hingegen an positiven und/ oder negativen Beispielen in einer Trainingsphase, welche Eigenschaften ein Zielkonzept charakterisieren, spricht man von überwachtem oder klassifizierendem Lernen [vgl. Frank & Witten 2001]. Entscheidend für einen Lernerfolg – gemessen als die Vorhersagegenauigkeit – ist dabei die Menge und die Qualität der Daten, aus denen Modelle abgeleitet werden. Jedes Beispiel, eine Instanz der Trainingsmenge, wird repräsentiert durch Attribut-Wert-Paare, die seine Merkmale beschreiben. Fehlerhafte oder fehlende

Attributwerte sowie für das zu lernende Zielkonzept irrelevante Attribute können zur Erstellung ungeeigneter Modelle führen, die später neue Testdaten falsch klassifizieren, z. B. aufgrund einer Überanpassung an die Trainingsdaten[73] [vgl. Frank & Witten 2001, 186ff]. Daher ist es wichtig, in einem ersten Schritt der Datensäuberung zuverlässige Daten und relevante Attribute zu selektieren, ggf. Attribute zusammenzuführen und Attributwerte zu normalisieren [vgl. Frank & Witten 2001, 52-60].

Weka ist ein an der Universität von Waikato entwickeltes System, das verschiedene induktive Lernverfahren, Methoden zur Datenaufbereitung und zur Evaluation der Lernergebnisse bietet [vgl. Frank & Witten 2000, 291]. Über eine graphische Benutzeroberfläche kann mit Datenfiltern und Lernalgorithmen experimentiert werden, die aber auch über die Kommandozeile aufgerufen oder mittels der Weka-Klassenbibliothek in ein eigenes Java-Programm eingebunden und weiterentwickelt werden können.

Um den erfolgreichen Einsatz maschinellen Lernens für das automatische Bereitstellen von Code-Beispielen zu untersuchen, werden verschiedene Lernalgorithmen in Weka experimentell auf ihre Fähigkeit überprüft, aus einer Menge von Programmbeispielen ein für den Code einer Problemsituation passendes Beispiel vorauszusagen. Damit handelt es sich um klassifizierendes Lernen, da die Beispielprogramme als Instanzen einer Trainingsmenge je eine eigene Klasse darstellen. Zu lernen ist, welche Code-Attribute – bspw. das Nutzen der Funktion 'indexOf' – ein spezielles Beispiel charakterisieren. Da jedes Beispiel eine eigene Klasse darstellt, findet keine Generalisierung von Mustern über Beispiele hinweg statt. Dies entspricht grundsätzlich dem Ansatz des instanzbasierten Lernens ('Nearest Neighbour'), bei dem wie beim fallbasierten Schließen die ähnlichste Instanz der Trainingsmenge ermittelt wird. Im Gegensatz zum fallbasierten Schließen kommt es jedoch zu keiner Adaption einer Lösung an die konkrete Testinstanz [vgl. Althoff 2001]. Es werden jedoch auch weitere Verfahren im Anschluss kontrastiv getestet und Ideen zum Ausbau des Ansatzes mithilfe solcher Verfahren entwickelt.

Um Weka mit den verschiedenen Lernalgorithmen zu nutzen, müssen die Beispielprogramme zunächst so aufbereitet werden, dass sie als Trainingsmenge dienen können. Wie die Java-Beispielprogramme dazu als Attribut-Wert-Paare repräsentiert und in das Weka-eigene Arff-Format überführt werden, beschreibt das folgende Kapitel.

---

[73] Ein gelerntes Konzept ist überangepasst, wenn ein anderes Konzept existiert, dass Testinstanzen fehlerfreier klassifizieren könnte, auch wenn es an der Trainingsmenge eine höhere Fehlerquote aufweist [vgl. Mitchell 1997, 67].

5.4.2 Implementierung

Da anhand des Quelltextes in einer aktuellen Problemsituation auf ein relevantes Beispielprogramm geschlossen werden soll, ist die Auswahl geeigneter Attribute zur Repräsentation wichtig. Zunächst schien es sinnvoll, jedes Java-Programm als einen abstrakten Syntax Baum (AST) darzustellen und Einheiten im Sinne von Produktionsregeln als Attribute zu nutzen, um auch die Struktur erfassen zu können[74]. Die Transformation in einen AST setzt jedoch syntaktische Korrektheit voraus, da nach den Regeln der zugrunde liegenden Grammatik[75] die einzelnen Tokens zu Einheiten wie einer 'for'-Schleife kombiniert werden. Von syntaktisch fehlerhaften Programmen kann (je nach Fehler und Grammatik) in Folge eines Parserkonflikts entweder kein AST gebildet oder die Attribute der korrekten Beispiele der Trainingsmenge können nicht abgeglichen werden.

In Anbetracht der häufigen und multiplen Syntaxfehler von Programmieranfängern, wurden daher einzelne Tokens von Programmen als Attribute gewählt. Alle Java-Symbole (ein Operator, ein Schlüsselwort, ein Klassenaufruf) wurden aus dem Quelltext extrahiert und beschrieben – unabhängig von der ursprünglichen Reihenfolge – als Attributsliste das Programm. Diese Attribute wurden vorerst nominal sowie numerisch erstellt, um später zu überprüfen, welcher Attributstyp zur Repräsentation geeigneter ist. Nominale Attribute sind kategorisch, so dass Attributsausprägungen einer der vorgegebenen – und als gleichwertig behandelten - Klassen entsprechen. In unserem Anwendungsfall wurden die Werte für ein Token '1' für vorhanden bzw. '0' für nicht vorhanden angenommen. Numerische Attribute hingegen geben Ausprägungen als Zahlenwerte abgestuft und somit ordinal skaliert an [Frank & Witten 2001, 52]. Ein Java-Programm wird folglich nicht nur dadurch charakterisiert, welche Tokens es enthält (nominal), sondern auch wie häufig die einzelnen Tokens auftreten.

Zur Extraktion der Attribute und einer Transformation in das Arff-Format von Weka wurde in Anlehnung an Bischoff et al. [2004] ein Programm mit vier Java-Klassen implementiert. Es stellt Funktionen für das Einlesen von Java-Programmen, das

---

[74] Diese Idee wurde inspiriert durch den Einsatz der AST-Repräsentation in ELP [Bancroft et al. 2004, 2005] und das Open-Source Tool PMD (http://pmd.sourceforge.net), in dem eine AST-Darstellung für die Prüfung auf Codekonventionen und Fehler genutzt wird.

[75] Eine selbst entwickelte Grammatik zum Parsen eines Quelltextes und ableiten von Attributen wie bspw. 'MethodDeclaration' oder 'VariableInitialisierung' könnte toleranter als der Java-Compiler gestaltet werden, Konflikte sind aber nicht vermeidbar. Die Entwicklung einer eigenen Grammatik mithilfe des Parsergenerators ANTLR beschreiben Krinke & Zeller [2004].

Löschen von Kommentaren, das Zerlegen des Quelltextes in Tokens, die Auswahl der Attribute und Ermittlung der jeweiligen Werte sowie für das Erzeugen einer Arff-Datei zur Verfügung. Für das Trainieren wurden zunächst über eine Datei, welche die Namen aller zu verarbeitenden Programme enthält, die Quelltexte eingelesen und Kommentare gelöscht. Zeilenkommentare wurden mithilfe regulärer Ausdrücke und Blockkommentare über direkten Stringvergleich entfernt. Um eine bestimmte Abstraktion zu erreichen wurden irrelevante Variationen in Programmen wie Variablennamen, Zahlen, Strings oder die Namen selbst erstellter Methoden standardisiert. Als aussagekräftige Attribute im Hinblick auf die semantische Ähnlichkeit gelten hingegen Schlüsselwörter, Methoden- und Klassenaufrufe der API und Operatoren.

Vor der Zerlegung des Codes in einzelne Tokens wurden Zeichenketten innerhalb von Anführungszeichen (als Werte von String-Variablen oder in Ausgabeanweisungen) durch 'einString' vereinheitlicht. Nach der Zerlegung des Quelltextes, bei der Tokens entlang von Zwischenräumen, Klammern, Punktzeichen und Operatoren getrennt wurden, war zudem die Vereinheitlichung von Variablen- und Methodennamen sowie Zahlen erforderlich. Durch den Abgleich mit einer Liste erfolgte das Ersetzen aller Tokens, die nicht einem Java-Operator, Punktzeichen, Klammer, Schlüsselwort oder Methoden- und Klassennamen der Bibliothek entsprachen. Zahlenwerte – abgeglichen über einen regulären Ausdruck – wurden unter dem Attribut 'eineZahl' und Bezeichner unter 'eineVariable' zusammengefasst. Um speziell auf Arrays und Methodenaufrufe besser eingehen zu können, wurden Tokens des Typs Bezeichner oder Datentyp bei nachfolgenden runden oder eckigen Klammern in ein Kompositumsattribut überführt. Da diese Reihenfolge in Folge von syntaktischen Fehlern im Testprogramm jedoch verletzt werden könnte, werden zudem die einzelnen Tokens als – dann teils redundante – Attribute genutzt.

Anhand der Liste aller so abgeleiteten Attribute der Trainingsbeispiele wurden die Werte eines Programms für jedes der Attribute ermittelt. Wurde dabei zu Programmstart 'nominal' als Attributtyp übergeben und kam ein Attribut in einem Quelltext mindestens einmal vor, galt der Wert '1', andernfalls der Wert '0'. Für den numerischen Attributtyp hingegen entsprach der Wert der Auftretenshäufigkeit eines Attributs in einem Programm. Die Attributsliste mit dem jeweiligen Attributtyp wurde als Kopf in eine generierte Arff-Datei und die Instanzen – repräsentiert durch die entsprechenden Werte – in den Datenbereich geschrieben. Das letzte Attribut der Trainingsinstanzen war dabei ihr Klassenname, der das zu lernende Zielkonzept angibt.

Zusätzlich erfolgt das Anlegen einer Textdatei mit allen Attributen, um Testprogramme gemäß den zugehörigen Trainingsbeispielen zu repräsentieren. Quelltexte, zu denen ein ähnliches Beispiel bestimmt werden sollte, wurden nach dem Einlesen dazu analog den Trainingsdaten aufbereitet. Auf das Löschen von Kommentaren, Ersetzen von syntaktischen Variationen der Bezeichner und Zahlen folgte der Abgleich der Tokens zur Attributsliste der Trainingsdaten. In einer erzeugten Arff-Datei über die Testinstanzen wurden die Programme als Instanzen durch ihre Werte für jedes Attribut dargestellt. Wiederum erfolgte zu Programmstart die Festlegung des Attributstyps (nominal oder numerisch). Der Klassenname als das letzte Attribut einer Testinstanz erhielt keinen Wert ('?'), da nach diesem Attribut klassifiziert werden sollte.

Für diese Klassifikation wurden unterschiedliche Lernalgorithmen des Weka-Pakets experimentell untersucht. Die Algorithmen, welche sich als grundsätzlich geeignet erwiesen, sollen vor einer ausführlichen Evaluation zunächst kurz vorgestellt werden.

5.4.3 Klassifikationsalgorithmen

Das in der Praxis für die Klassifizierung von Texten [vgl. Joachims 2002] oft erfolgreich genutzte Verfahren **Naïve Bayes** [Frank & Witten 2001, 88-92] bedient sich der statistischen Modellierung von Wahrscheinlichkeiten nach Bayes. Unter der (naiven) Annahme, dass alle Attribute unabhängig voneinander und gleich relevant sind, werden anhand der Trainingsdaten die Wahrscheinlichkeiten geschätzt, mit der ein bestimmter Wert eines Attributs auf eine Klasse weist. Dazu werden im Training die Häufigkeitsverteilungen der Attributwerte für jede Klasse – als 'a posteriori Wahrscheinlichkeiten' – ermittelt. Die Gesamtwahrscheinlichkeit der Zugehörigkeit einer Instanz zu einer Klasse entspricht dem Produkt der Einzelwahrscheinlichkeiten seiner Attributswerte für eine Klasse[76]. Dabei können redundante und irrelevante Attribute die Verhältnisse verzerren, da ihre Häufigkeitsverteilungen als unabhängige und gleichwertige Auftretenswahrscheinlichkeiten in die Berechnung eingehen. Der Trainingserfolg hängt umso stärker von einer großen und repräsentativen Datenmenge ab. Das Verfahren ist jedoch einfach, schnell und robust bezüglich fehlender Werte [Frank & Witten 2001, 88-92].

Instanzbasierte Verfahren (IBK) [Frank & Witten 2001, 76-80, 123ff; Mitchell 1997, 230-246] nehmen als einzige Lerntechniken keine explizite Wissensmodellierung in

[76] Voraussetzung dafür ist die Annahme der Unabhängigkeit der Attribute, da die bedingte Wahrscheinlichkeit somit zum Produkt der Einzelwahrscheinlichkeiten vereinfacht werden kann.

Form von Mustern oder Modellen vor; sie speichern Trainingsinstanzen 'lediglich' ab. Lernen findet erst statt, wenn eine neue Instanz klassifiziert wird, indem sie die Klasse der ähnlichsten gespeicherten Instanz erhält. Das Maß für die Ähnlichkeit ist dabei die Euklidische Distanz, die Entfernung der Instanzen in einem n-dimensionalen Raum, wobei jede Dimension ein Attribut repräsentiert. Bei nominalen Attributen kann damit nur zwischen '0' bei Gleichheit und '1' bei Ungleichheit entlang einer Dimension differenziert werden [vgl. Frank & Witten 2001, 77]. Hingegen erlauben numerische Werte eine feiner abgestimmte Distanzmessung. Zur instanzbasierten Klassifikation werden in der Praxis häufig mehrere nächste Nachbarn ('k-Nearest Neighbor') verwendet, um fehlerhafte ('noisy') Daten besser abzufangen. Fehlende Werte und irrelevante Attribute sind im Hinblick auf die Distanzberechnung kritisch, wobei im folgenden Anwendungsfall nur irrelevante Attribute Einfluss haben. Gewichtungsmöglichkeiten für einzelne Attribute und eine geeignete Distanzmetrik generell sind daher das Kernproblem des Verfahrens [Frank & Witten 2001, 123ff]

In Anlehnung an die Informationsverarbeitung im menschlichen Gehirn bestehen **Neuronale Netze** ('MultilayerPerceptron'**)** aus einzelnen Prozessoren oder 'Units', die untereinander in einem Netzwerk verbunden sind [vgl. Mitchell 1997, 82]. In einem MultilayerPerceptron sind diese Units in mehreren Schichten angeordnet, wobei die erste Schicht als Eingabe die Attributwerte der Instanzen erhält, die Ausgaben entsprechen der letzten Schicht den Zielkonzepten.

> „Ein Prozessor im neuronalen Netz wird aktiviert, wenn die Summe aller hereinkommenden Signale einen bestimmten Schwellenwert überschreitet, wobei den eingehenden Signalen an jedem Prozessor unterschiedliche Gewichtungsfaktoren zugeordnet sind.“[Brockhaus 2003].

Im Training werden innerhalb des Netzes die Gewichte der Neuronenfunktionen – die Verbindungsstärke zwischen den Neuronen – von einem anfänglichen Zufallswert in mehreren Iterationen über die Instanzen so angepasst, dass Vorhersagefehler für die Trainingsdaten minimiert werden. Dabei werden mittels 'Backpropagation' Ausgabefehler an das Netz zurückgemeldet und die Gewichte daraufhin verändert [vgl. Mitchell 1997, 97f]. Neuronale Netze eignen sich für Probleme mit komplexen Entscheidungsgrenzen, da durch die vielschichtigen Verknüpfungen unterschiedlichste Beziehungen zwischen Attributen modelliert werden können [vgl. Mitchell 1997, 81-87]. Lange Trainingszeiten, eine schlechte Nachvollziehbarkeit des gelernten Modells und die Gefahr der Überanpassung an die Trainingsmenge, sind mögliche Nachteile des Verfahrens [vgl. Mitchell 1997, 108-111]. Um Letzteres zu vermeiden, gilt es, die besten

Parameter für die Zahl der Iterationen, die Lernrate (Grad der Gewichtsänderungen) an einer Testmenge zu bestimmen [vgl. Mitchell 1997, 108-111].

Auch **Support-Vektor-Maschinen** (SVM) können komplexe, nicht-lineare Entscheidungsgrenzen abbilden, indem Trainingsdaten als Punktvektoren in einen höherdimensionalen Raum transformiert werden, so dass eine lineare Grenze zwischen zwei Konzeptklassen entsteht [Christianini & Sahwe-Taylor 2001, 27ff]. Lernen entspricht hier der Bestimmung der Gewichte für diese Transformation. Da mehrere lineare Entscheidungsgrenzen denkbar sind, wird anhand von Stützvektoren die mit größtem Abstand trennende Entscheidungsgrenze, die 'Maximum Margin Hyperplane', ermittelt. Diese liegt zwischen den Instanzen unterschiedlicher Klassen, die sich am nächsten sind – den Support-Vektoren [vgl. Frank & Witten 2001, 204-210]. Da hier ein Zweiklassenproblem vorliegt, müssen für die Anwendung auf Daten mit vielen Zielklassen mehrere binäre SVMs trainiert werden [vgl. Christianini & Sahwe-Taylor 2001]. Das Verfahren bietet generell eine sehr genaue Klassifikation und ist robust gegenüber Fehlern in Trainingsdaten sowie gegenüber Überanpassung. Jedoch ist es sehr zeit- und rechenintensiv [vgl. Frank & Witten 2001, 204-210].

5.4.2 Technische Evaluation

In der technischen Evaluation wird die Angemessenheit verschiedener Lernverfahren für das Abrufen zu einer Problemsituation ähnlicher Beispielprogramme untersucht. Zur Bewertung der Leistung eines Klassifikationsverfahrens für eine bestimmte Datenmenge existieren dabei unterschiedliche Kriterien [vgl. Frank & Witten 2001, 128-136]. In Weka wird während des Trainierens die Fehlerrate des Lernverfahrens bezüglich der korrekten Klassifikation errechnet. Dazu sagt der Klassifizierer anhand des gelernten Modells die Klasse einer jeden Instanz der Trainingsmenge voraus, so dass die Fehlerrate aufzeigt, wie viele Fehler bei der Zuordnung entstanden. Diese Fehlerrate ist zumeist jedoch kein sinnvoller Indikator für die Leistung eines Klassifizierers für neue Daten, da in Folge einer möglichen Überanpassung an die Trainingsdaten aus der Fehlerrate nicht auf die Generalisierbarkeit des Gelernten geschlossen werden kann[77] [vgl. Frank & Witten 2001, 128ff]. Auch hier ist vielmehr entscheidend, wie adäquat

[77] Ist ein Konzept zu genau auf die Trainingsdaten abgestimmt, können unrepräsentative oder fehlerhafte Instanzen bewirken, dass später 'gute' Beispiele durch 'schlechte' Regeln falsch klassifiziert werden. [vgl. Frank & Witten 2001, 128ff]

sich das Modell für die Voraussage eines relevanten Beispielprogramms zu den Testprogrammen der Anfänger erweist.

Für das Evaluieren an einer Testmenge liegen jedoch häufig nicht genügend Daten vor, da schließlich möglichst alle verfügbaren Daten für das Training eingesetzt werden sollen, damit ein generalisierbares Konzept gelernt wird [vgl. Frank & Witten 2001, 130-135]. Die Methode der Kreuzvalidierung begegnet diesem Problem damit, dass die Trainingsdaten auch gleichzeitig als Testdaten verwendet werden. Dabei wird die Trainingsmenge in Partitionen unterteilt (bspw. zehn) und nacheinander jede Partition als Testmenge zurückgehalten, während das Modell an den Restlichen gelernt wird. Um Effekten einer zufälligen Unterteilung entgegenzuwirken, wird dieser Vorgang mehrmals wiederholt [vgl. Frank & Witten 2001, 133ff]. Auch die Kreuzvalidierung ist in unserem Anwendungsfall aufgrund des grundsätzlich instanzbasierten Ansatzes, indem jede Instanz eine eigene Klasse bildet, eine unpassende Evaluationsmethode. Denn zwangsläufig erhalten die zurückgehaltenen Instanzen die Klasse einer anderen Instanz, da ihre eigene im Training nicht berücksichtigt wurde.

Aus diesem Grund wird als Evaluationskriterium die Fehlerrate an einer kleinen Testmenge gewählt, wobei als Testinstanzen die Code-Fragmente der Programmieranfänger in den Problemsituationen des WOz-Test genutzt werden. Dabei wurden nur solche Code-Beispiele ausgewählt, bei denen der Tutor nicht auf Anfrage der Lernenden, sondern aufgrund einer im Code erkennbaren Problemsituation selbstständig ein Beispiel als Hilfestellung gab. Von den ursprünglich 37 Quell-Codes konnten daher nur 29 zur technischen Evaluierung herangezogen werden. Diese stammen aus 15 Kooperationssituationen[78]. Für die Bewertung der Relevanz der gelieferten Beispiele ist zu beachten, dass die Erwartungen der Gruppenmitglieder nicht immer den vom Tutor erkannten Problemen entsprachen. Dies war v. a. dann der Fall, wenn die Schwierigkeit in der weiteren Entwurfsplanung, statt im aktuellen syntaktischen, semantischen oder logischen Fehler bestand. Aus diesem Grund wurden die von den Klassifizierern zugeordneten Klassen zudem intellektuell nach Art einer Expertenevaluation mit den vom Tutor erbrachten verglichen und dabei auf syntaktische und semantische Ähnlichkeiten hin überprüft. So wurde die Möglichkeit mit eingeschlossen, dass auch ein differierendes Beispiel aus der automatischen Erkennung relevant für die weitere Aufgabenbearbeitung sein könnte.

[78] Dieses Verhältnis von Quellcodes zu Problemsituationen ergibt sich aus der Tatsache, dass in zwei Benutzertests alle drei Teammitglieder jeweils in einem eigenen Dokument gearbeitet haben.

Begründet in den genannten Schwierigkeiten bei der Erkennung von Entwurfsproblemen im Programm-Code und der sehr geringen Datenmenge können die Vorteile und Schwächen der getesteten Klassifizierer im Folgenden lediglich exemplarisch dargestellt werden.

Der Naïve Bayes-Klassifizierer mit nominalen Attributen stimmt zwar nur in zwei Fällen mit dem Beispiel des WOz-Test überein, jedoch gibt er einige Beispiele, die dennoch als relevant für die Lösung des aktuellen Aufgabenschrittes angesehen werden können. Der Test mit numerischen Attributen ergibt keine direkten Übereinstimmungen. Zudem werden in einigen Fällen relevante und vom Tutorenvorschlag differierende Beispiele nicht, wie unter Anwendung nominaler Attribute, erkannt: Für die Problemsituation in einem Quelltext, der neben der Klassendeklaration, 'Main'-Methode und einer String-Initialisierung die Importanweisung für 'StringTokenizer' enthält, wird durch den Naïve Bayes-Klassifizierer mit nominalen Attributen der Intention des Programmierers entsprechend ein Beispiel vorausgesagt, dass die Implementierung eines 'StringTokenizers' zeigt. Dieser Fall tritt bei der Verwendung numerischer Attribute nicht auf, stattdessen wird der Einsatz des Bedingungs-Operators dargestellt. Die in der nominalen Variante vorausgesagten relevanten Beispiele sind durchgehend in ihrer Komplexität, Syntax und Semantik exakter auf die Anforderung des aktuellen Aufgabenschrittes abgestimmt.

Während der Einsatz des IBk-Klassifizierers mit nominalen Attributen dieselben Ergebnisse erbringt wie der Naïve Bayes-Klassifizierer mit eben diesen Eigenschaften, treten bei der Verwendung numerischer Attribute Unterschiede auf. Ersterer zeigt in der numerischen Variante zumindest zwei Übereinstimmungen, Naïve Bayes hingegen erweist sich mit numerischen Eigenschaften als weniger geeignet für die Zuordnung der Problembeispiele. Hier wird deutlich, dass die Verwendung nominaler Attribute im Allgemeinen besser geeignet ist, da das häufige Auftreten bestimmter javaspezifischer Ausdrücke wie 'String' im Falle numerischer Attribute die Auswertung und damit die Zuordnung der Problemfälle verzerrt. Auch für die Support-Vektor-Maschine und das MultilayerPerceptron ist diese Schwierigkeit festzustellen. Beide Verfahren haben für nominale Attribute teils sehr passende teils weniger ähnliche Beispiele vorausgesagt. In Anbetracht des fehlenden Relevanz-Feedbacks und der geringen Datenmenge kann jedoch keine eindeutige Beurteilung der besonderen Güte eines speziellen Klassifizieres vorgenommen werden. Jedes Verfahren scheint für bestimmte Attributwert-Klassenkombination besser als für andere zu arbeiten. Die experimentelle Einstellung der

Parameter wie bspw. der Lernrate oder Distanz ließ für alle Algorithmen kaum veränderte Voraussagen beobachten. Interessant ist, dass zumeist soweit ähnliche Beispiele bestimmt werden können, dass die Kosten für eine falsche Klassifikation gering sein dürften [vgl. Frank & Witten 2001, 146ff]. Weiterzuverfolgen wäre auch das Clustern [vgl. Frank & Witten 2001, 80f] ähnlicher Beispiele zum Lernen javaspezifischer Konzepte, um über Beispiele hinweg die Generalisierungsfähigkeit zu überprüfen.

Ein großer Nachteil, den alle Klassifizierer mit nominalen Attributen v. a. bei kleinen Code-Fragmenten bspw. mit nur einer String-Initialisierung zeigen, ist die undifferenzierte Zuordnung zu einem Beispiel, in einem solchen Fall zur Beispielklasse 'HelloWorld.java'. Als abschließender Eindruck zu den getesteten Lernverfahren lässt sich sagen, dass alle vier Techniken als prinzipiell geeignet scheinen und erst eine systematische Evaluierung anhand größerer Daten mit Bewertung durch Programmieranfänger (und bspw. der Erhebung von Maßen wie 'Recall' und 'Precision') über die Auswahl eines Verfahrens oder den Einsatz eines Metaklassifizierers entscheiden kann. Zur Verbesserung scheinen zusätzlich einerseits Techniken zur vorgeschalteten Attributselektion (ähnlich der Ermittlung des 'information gain'), wie sie auch in Weka bereitstehen, andererseits auch die Exploration von (Code-)Retrieval-Verfahren mit Ähnlichkeitsmaßen wie Vektorraum-Modell sinnvoll.

6 FAZIT

Das Lernen einer objektorientierten Programmiersprache wie Java stellt für Anfänger oft eine Herausforderung dar. In der empirischen Untersuchung virtueller Teams ließen sich die in der psychologischen Literatur berichteten typischen Anfängerfehler und -probleme ermitteln, wobei aufgrund des Datencharakters nicht auf die zugrunde liegenden Ursachen geschlossen werden kann. Häufige Syntax- und Semantikfehler, die zumeist in Kombination mit Schwierigkeiten bezüglich des algorithmischen Designs von Programmen auftreten, konnten so identifiziert werden. Die Kategorisierung von Fehlern und Problemsituationen in Syntax, Semantik und Logik anhand der Java-Sprachspezifikation erwies sich insofern als sinnvoll, als somit für die automatische Erkennung erste Hinweise zur Verfügung stehen. Wenden Anfänger in Problemsituationen wissensarme Techniken wie 'Versuch-und-Irrtum' an, indem sie bspw. Stellen im Quell-Code häufig ändern und rekompilieren, können die wiederholten Compiler-Fehler auf ein syntaktisches Problem deuten.

Ebenso wie auf Laufzeitfehler könnte ein elektronischer Tutor z. B. mit Hilfestellung durch das Präsentieren eines Beispiels reagieren. Generell hat die Analyse aber auch gezeigt, dass aufgrund der kollaborativen Lernsituation – effektive Kommunikation und aktiven Wissenstransfer vorausgesetzt – Teams gerade syntaktische und semantische Fehler oft zeitnah eigenständig beheben können.

Aufgrund der Beschränkung auf fachliche Problemsituationen konnten in den Beobachtungen spezifische Probleme mit der Planung und Spezifikation von Programmen nicht erfasst werden. Unter systematischer Einbeziehung der Kodierung und der Kommunikation könnten die Ergebnisse dieser Untersuchung mit denen von Göldner [2005] zusammengeführt und an weiteren Daten überprüft werden. Derartige Probleme bei der Gestaltung der Lösung können mithilfe des vorgeschlagenen Ähnlichkeitsvergleichs von vorgefertigten Beispielprogrammen zum Quell-Code jedoch nicht zur Bereitstellung geeigneter Beispiele genutzt werden.

Die maschinellen Lernverfahren scheinen für die Bereitstellung von Beispielen zur Überwindung syntaktischer oder semantischer Probleme durchaus einsetzbar. Die Evaluierung möglicher Klassifizierer und ihrer Anwendbarkeit auf den speziellen Fall der Programmierarbeit im Projekt VitaminL muss dazu jedoch anhand einer ausreichend großen Datenbasis mit einem breiteren Aufgabenspektrum differenziert durchgeführt werden. Die Möglichkeit, ein fehlerhaftes Code-Fragment isoliert vom restlichen Code als Grundlage der Zuordnung eines relevanten Beispieles bspw. durch Markierung im

Editor angeben zu können, würde dabei sicher zu einer Verbesserung der Klassifizierungsergebnisse führen. Denn auf diese Weise würden für diese konkrete Situation irrelevante Attribute nicht in den Ähnlichkeitsvergleich einbezogen. Dazu müssten die Lernenden jedoch eben dieses Code-Fragment identifizieren können.

Ihr Potential verdeutlichen aufgabenrelevante Beispiele besonders zur korrekten und effizienten Entwurfsgestaltung, dafür müsste zumindest eine Referenzlösung vorliegen. Die Erfassung und Klassifizierung dieser Problemsituationen ist jedoch nicht anhand des reinen Programm-Codes möglich. Vielmehr müssen hier weitere Indikatoren wie die Zeiten zwischen Programmieraktivitäten (z. B. lange Pausen) und v. a. die Auswertung der begleitenden Chatkommunikation herangezogen werden. Zusätzlich könnte überwachtes Lernen definierter Konzepte zur Bewältigung von Aufgabenteilschritten eine Möglichkeit zur Unterstützung planungsbezogener Problemsituationen bieten. Vorstellbar wäre auch die Verwendung von Clustering-Verfahren, um Gruppen ähnlicher Beispiele zu identifizieren, die dann bspw. als Indexe für Klassen eines überwachten Lernens zu nutzen wären. Ein Abgleich solcher erlernten Konzepte mit den Strukturen bereits programmierter Teillösungen der virtuellen Teams kann als Grundlage für die Bereitstellung weiterführender Entwurfsvorschläge dienen.

Weiterhin ist die Erstellung einer indizierten Beispielsbibliothek denkbar, die über eine Stichwortsuche selbstständig von den Lernenden zur Unterstützung der Aufgabenbearbeitung herangezogen werden kann. Die Einschränkung der Verfügbarkeit der Beispielsfälle, angepasst auf die jeweiligen Aufgabenstellungen, könnte in diesem Zusammenhang die gezielte Vermittlung bestimmter Konzepte und Verfahren unterstützen. Dieser Ansatz wird durch die Ergebnisse der offenen Benutzerbefragung unterstützt, nach denen die Einbindung der Java-API von den Studenten bei der Bearbeitung der Programmieraufgaben als sehr hilfreich eingeschätzt wurde.

Trotz des positiven Feedbacks zur beispielbasierten Unterstützung aus den Simulationstests muss dabei jedoch bedacht werden, dass Beispiele ähnlich Techniken des automatischen Debuggens nur eine bestimmte Reichweite haben. Nicht immer gelingt der Transfer aus den Beispielen, auch wenn die korrekte Lösung in ihnen abgebildet ist. Als eine mögliche unterrepräsentierte Rolle, die des Entdeckers [vgl. Kölle & Langemeier 2004a], erfüllen sie aber gerade für das algorithmische Design ihre unterstützende Funktion.

Zusammenfassend lässt sich sagen, dass der Ansatz einer beispielbasierten Unterstützung bei der Programmierung objektorientierter Software durch virtuelle Teams in den erfassten und klassifizierten Problemsituationen Erfolg versprechend ist. Die Zuordnung relevanter Beispiele muss jedoch für die verschiedenen Klassen nach unterschiedlichen Kriterien und anhand spezifischer Verfahren erfolgen. Die Einbeziehung weiterer Faktoren, wie der Kommunikation zwischen den Gruppenteilnehmern, ist für die Weiterentwicklung des Ansatzes unbedingt notwendig. Vor allem aber ist die Nützlichkeit des Ansatzes auch für Anfänger in den ersten Wochen der Programmierung zu evaluieren, da hier die Gewichte bezüglich der Bedeutung bestimmter Problemtypen sich verschieben dürften.

LITERATURVERZEICHNIS

ADAM, A. & LAURENT, J. P. (1980): LAURA: A system to debug student programs. In: Journal of Artificial Intelligence. 15, S. 75-122.

ADAMS, E. S., FITZGERALD, S., FONE, W., HAMER, J., LISTER, R., LINDHOLM, M., MCCARTNEY, R., MOSTRÖM, J. E., SANDERS, K., SEPPALA, O., SIMON, B. & THOMAS, L. (2004): A multi-national study of reading and tracing skills in novice programmers. In: ACM SIGCSE Bulletin, 36 (4).

ALA-MUTKA, K., JÄRVINEN, H.-M. & LAHTINEN, E. (2005): A Study of the Difficulties of Novice Programmers. In: Proceedings of ITiCSE '05. 27.-29. Juni, 2005, Monte de Caparica, Portugal. ACM. S. 14-18.

ALJUNID, S. A., NORDIN, M. J., SHUKUR, Z. & ZIN, A. M: A Knowledge-based Automated Debugger in Learning System. http://arxiv.org/PS_cache/cs/pdf/0101/0101008.pdf. (Verifizierungsdatum 18.10.2005)

ALLEN, E., CARTWRIGHT, R. & STOLER, B. (2001): DrJava: A lightweight pedagogic environment for Java. http://drjava.sourceforge.net/papers/drjava-paper.shtml. (Verifizierungsdatum 18.10.2005)

ALMAZAN, C. B., FOSTER, J. S. & RUTAR, N. (2004): A Comparison of Bug Finding Tools for Java. In: Proceedings of the 15th IEEE International Symposium on Software Reliability Engineering (ISSRE'04). S. 245-256. www.cs.umd.edu/~jfoster/papers/issre04.pdf. (Verifizierungsdatum 18.10.2005)

ALTHOFF, K.-D. (2001): Case-Based Reasoning. In: S. K. Chang (Hrsg.): Handbook on Software Engineering and Knowledge Engineering. Vol. 1 "Fundamentals", World Scientific, S. 549-588. ftp://cs.pitt.edu/chang/handbook/23.pdf. (Verifizierungsdatum 18.10.2005)

ANDERSON, J.R. (2000[5]): Cognitive Psychology and Its Implications. New York: Worth Publishers.

ANDERSON, J.R., BOYLE, C.F., CORBETT, A.T. & LEWIS, M. W. (1990): Cognitive Modeling and Intelligent Tutoring. In: Clancey, W.J. & Soloway, E. (1990): Artificial Intelligence and Learning Environments. Cambridge & London: MIT Press, Amsterdam: Elsevier Science Publishers. S. 7-49.

ANDERSON, J.R. & CORBETT, A.T. (2001): Locus of Feedback Control in Computer-Based Tutoring: Impact on Learning Rate, Achievement and Attitudes. In: Proceedings of the SIG-CHI on Human factors in computing systems, 31. März - 4. April, 2001, Seattle, WA, USA, ACM, 2001. 3 (1). S. 245-252.

ANDERSON, J.R. & REISER, B.J. (1985): The LISP Tutor. In: Byte. 10 (4), S.159-175.

ANDERSON, J. R. & SCHOOLER, L. J. (1990): The disruptive potential of immediate feedback. In: Proceedings of the Twelfth Annual Conference of the Cognitive Science Society. Cambridge, MA. Hillsdale, NJ: Cognitive Science Society. S. 702-708.

ANDERSON, J. R. & SWARECKI, E. (1986): The Automated Tutoring of Introductory Computer Programming. In: Communications of the ACM. 29 (9), S. 842-849.

BALZERT, H. (1999): Lehrbuch Grundlagen der Informatik. Konzepte, Notationen in UML, Java, C++, Algorithmik und Softwaretechnik. Heidelberg und Berlin: Spektrum Akademischer Verlag.

BALZERT, H. & BALZERT, H. (2004): Modellieren oder Programmieren oder beides? In: LOG IN. 128/129 (2004), S.20-25.

BANCROFT, P., ROE, P. & TRUONG, N. (2004): Static Analysis of Students' Java Programs. In: Lister, R. & Young, A. (Hrsg.): Sixth Australasian Computing Education Conference (ACE 2004), Dunedin, New Zealand, 18.-22. Januar, 2004, CRPIT 30 Australian Computer Society 2004, S. 317-325.

BANCROFT, P., ROE, P. & TRUONG, N. (2005): Automated Feedback for "Fill in the Gap" Programming Exercises. In: Young, A. & Tolhurst, D. (Hrsg.): Seventh Australasian Computing Education Conference (ACE 2005), Newcastle, NSW, Australia, Januar/Februar 2005. CRPIT 42 Australian Computer Society 2005. S. 117-126.

BAUKNECHT, K. (2000): Systemklasse: gemeinsame Informationsräume. Vorlesungsunterlagen Computerunterstützte Gruppenarbeit. Institut für Informatik der Universität Zürich. http://www.ifi.unizh.ch/ikm/Vorlesungen/CSCW/CSCW_WS0001/CSCW_WS0001/Documents/GemInformationsraeume.pdf. (Verifizierungsdatum 18.10.2005)

BAYMAN, P. & MAYER, R. E. (1983): A Diagnosis of Beginning Programmers' Misconceptions of BASIC Programming Statements. In: Communications of the ACM. 26 (9), S. 677-679.

BASSOK M., CHI, M. T. H., GLASER R., LEWIS M.W. & REIMANN P. (1989). Self-Explanations: How Students Study and Use Examples in Learning to Solve Problems. In: Cognitive Science, 13/ 1989, S. 145-182.

BEN-ARI, M. (2001): Program visualisation in theory and practice. In: Upgrade. 11(2), S. 8-11.

BEN-ARI, M. (1998): Constructivism in Computer Science Education. In: Proceedings of SIGSE ´98. ACM, S. 257-261.

BERGE, O., BORGE, R. E., FJUK, A., HOLMBOE, C., KAASBØLL, J., & SAMUELSEN, T. (2004): Learning Object-Oriented Programming. In: Proceedings of PPIG 2004, 16th workshop PPIG online.

BERNAT, A., CARSWELL, L., DEVEAUX, D., ELLIS, A., FRISON, P., MEISALO, V., MEYER, J., NULDEN, U., RUGELJ, J. & TARHIO, J.(1998): Resources, Tools, and Techniques for Problem Based Learning in Computing. SIGCUE outlook, 26, S. 41-56.

BISCHOFF, K., DOPATKA, A., GÖLDNER, A. & KRAMER, A. (2004): Semi-automatische Domänenerweiterung. Unveröffentlichter Projektbericht Seminar Semantik Web und Ontologien. Universität Hildesheim.

BISHOP, A., COLLINS, J., COOKE, J., GREER, J., KUMAR, V., MCCALLA, G. & VASSILEVA, J. (1998): The Intelligent Helpdesk: Supporting Peer-Help in a University Course. In: Intelligent Tutoring Systems. S. 494-503.

BONAR, J. & CUNNINGHAM, R. (1988): Bridge: An intelligent tutor for thinking about programming. In: Self, J. (Hrsg.): Artificial Intelligence and human learning: Intelligent computer aided instruction. London: Chapman & Hall, S. 391-409.

BONAR, J. & SOLOWAY, E. (1989): Preprogramming Knowledge: A Major Source of Misconceptions in Novice Programmers. In: Soloway & Spohrer (1989), S. 325-354.

BORTZ, J. & DÖRING, N. (1995[2]): Forschungsmethoden und Evaluation (für Sozialwissenschaftler). Berlin et al.: Springer-Verlag.

BROCKHAUS (2003): Computer und Informationstechnologie. Leipzig, Mannheim: F.A. Brockhaus.

BRUCKMAN, A. & EDWARDS, E. (1999): Should We Leverage Natural-Language Knowledge? An Analysis of User Errors in a Natural-Language-Style Programming Language. In: Proceedings CHI'99. 15-20. Mai, 1999 Pittsburgh PA USA, S. 207-214.

BRUN, Y. & ERNST, M. D. (2004): Finding latent code errors via machine learning over program executions. In: Proceedings of the 26th IEEE International Conference on Software Engineering (ICSE'04). S. 480-490.

BRUSILOVSKY, P. (1999): Adaptive and Intelligent Technologies for Web-based Education. In: C. Rollinger and C. Peylo (Hrsg.): Special Issue on Intelligent Systems and Teleteaching, Künstliche Intelligenz, 4, S. 19-25.

BRUSILOVSKY, P. & PEYLO, C. (2003): Adaptive and Intelligent Web-based Educational Systems. In: International Journal of Artificial Intelligence in Education. 13, S. 156-169. IOS Press.

BRUSILOVSKY, P., SCHWARZ, E. & WEBER, G. (1996): ELM-ART: An Intelligent Tutoring System on World Wide Web. In: Frasson, C., Gauthier, G. & Lesgold, A. (Hrsg.): Intelligent Tutoring Systems, Proceedings of the third International Conference, ITS '96, Montréal, Canada, 12.-14. Juni, 1996, LNCS 1086, Springer, S. 261-269. http://www.sis.pitt.edu/~peterb/papers/ITS96.pdf (Verifizierungsdatum 18.10.2005)

BRUSILOVSKY, P., SPECHT, M. & WEBER, G. (1995): Towards Adaptive Learning Environments. In: Proceedings der 25. GI-Jahrestagung und 13. Schweizer Informatikertag GISI'95, Zürich, 18-20. September 1995. S. 322-329. http://www2.sis.pitt.edu/~peterb/papers/gisi95.pdf. (Verifizierungsdatum 18.10. 2005)

BRUSILOVSKY, P. & WEBER, G. (2001): ELM-ART: An Adaptive Versatile System for Web-based Instruction. In: International Journal of Artificial Intelligence in Education. 12, S. 351-384

CERECKE, C. (2001): Repairing syntax errors in LR-based parsers. In: Proceedings of the 25th Australasian Computer Science Conference (ACSC 2002), Melbourne, Australia, S.17-22.

CHRISTIANINI, N. & SAHWE-TAYLOR, J. (2001[2]): An Introduction to Support Vector Machines and other kernel-based learning methods. Cambridge: Cambridge University Press.

CLEVE, H. & ZELLER, A. (2001): Automatisches Debugging. Universität Passau: Skript zum Hauptseminar.

CONTRERAS, J., FAVELA, J., PRIETO, M. & VIZCAINO, A. (2000): An Adaptive, Collaborative Environment to Develop Good Habits in Programming. In: Frasson, C., Gauthier, G. & VanLehn, K. (2000) (Hrsg.): ITS 2000. LNCS 1839, Springer, S. 262-271.

CORBETT, J. C., DWYER, M. B., HATCLIFF, J., LAUBACH, S., PASAREANU, C. S., ROBBY & ZHENG, H. (2000): Bandera: Extracting Finite-state Models from Java Source

Code. In: Proceedings of the 22nd International Conference on Software Engineering, Limerick Ireland, S. 439–448.

CRUZ, M. & SISON, R. (2002): From PROLOG to JAVA: Applying MEDD to Object Oriented Programming. In: Proceedings of the International Conference on Computers in Education (ICCE'02). IEEE.

DAVIES, S. P. (1993): Models and theories of programming strategy. In: International Journal of Man-Machine Studies, 39, S. 237–267.

DEEK, F.P. & MCHUGH, J.A. (1998): A Survey and Critical Analysis of Tools for Learning Programming. In: Computer Science Education. 8 (2), S. 130-178.

DÉTIENNE, F. (1997): Assessing the cognitive consequences of the object-oriented approach: A survey of empirical research on object-oriented design by individuals and teams. In: Interacting with Computers. 9, S. 47-72.

DIECKMANN, A. (1995): Empirische Sozialforschung. Grundlagen, Methoden, Anwendungen. Reinbeck bei Hamburg: Rohwolt Taschenbuch Verlag GmbH.

DILLENBOURG, P. & SELF, J. (1992): A Framework for Learner Modelling. In: Interactive Learning Environments. 2 (2), S. 111 - 137. Online. http://tecfa. unige.ch/tecfa/publicat/dil-papers-2/Dil.7.2.4.pdf. (Verifizierungsdatum 18.10. 2005)

DOMINGUE, J. & MULHOLLAND, P. (o. J.): Teaching Programming at a Distance: The Internet Software Visualization Laboratory. http://www-jime.open.ac.uk/97/ 1/isvl-01.html. (Verifizierungsdatum 18.10.2005).

DRJAVA: http://www.drjava.org; http://drjava.sourceforge.net/papers/drjava-paper.shtml. (Verifizierungsdatum 18.10.2005).

DU BOULAY, B. & VIZCAINO, A. (2002): Using a Simulated Student to Repair Difficulties in Collaborative Learning. In: Proceedings of the International Conference on Computers in Education, ICCE 2002, 3.-6. Dezember, 2002, Auckland, New Zealand. IEEE Computer Society, S. 349-353. http://www.cogs.susx.ac.uk/users/bend/papers/icce2002.pdf. (Verifizierungsdatum 18.10.2005)

DU BOULAY, B. (1989): Some difficulties in learning to program. In: Soloway & Spohrer (1989), S. 283-299.

DU BOULAY, B., O'SHEA , T., & MONK, J. (1989): The Black Box Inside the Glass Box: Presenting Computing Concepts to Novices. In: Soloway & Spohrer (1989), S. 431-446.

DUCASSÉ, M. & EMDE, A.-M. (1988): A Review of Automated Debugging Systems: Knowledge, Strategies and Techniques. In: Proceedings of the 10th International Conference on Software Engineering. IEEE Computer Society Press, S. 162-171.

DYCK, J. L., MAYER, R. E., VILBERG, W. (1986): Learning to program and learning to think: What's the connection? In: Communications of the ACM. 29 (7), S. 605-610.

DYNAMICJAVA: http://koala.ilog.fr/djava/#differences. (Verifizierungsdatum 18.10.2005)

EBRAHIMI, A. (1994): Novice programmer errors: language constructs and plan compostion. In: International Journal of Human-Computer Studies. 41, S. 457-480.

ECLIPSE: Eclipse 3.1 Documentation Java Development User Guide. http://www.eclipse.org/downloads/index.php. (Verifizierungsdatum 18.10.2005)

EDELMANN, WALTER (2000[6]): Lernpsychologie. Weinheim: Beltz, Psychologie Verlags Union.

FARRELL, R. & SAUER, R. (1982): GRAPES user's manual. Carnegie-Mellon University Pittsburgh, Department of Psychology.

FIX, V., SCHOLTZ, J. & WIEDENBECK, S. (1993): Mental Representations of Programs by Novices and Experts. In: Proceedings Interchi 1993, 24.-29. April, 1993, ACM, S. 74-79.

FLANAGAN, C., LEINO, K. R. M., LILIBRIDGE, M., NELSON, G., SAXE, J. B. & STATA, R. (2002): Extended Static Checking for Java. In: Proceedings PLDI'02, 17.-19. Juni, 2002, Berlin, ACM, S. 234-245.

FLEURY, A. E. (2000): Programming in Java: Student-Constructed Rules. Proceedings of the 31st SIGCSE, (2000), ACM press, S. 197-201.

FOX, R. & UTI, N.V. (2003): Automated Debugging of Java Syntax Errors Through Diagnosis. In: Norse Scientist. 1, Northern Kentucky University Press, S. 73-81.

FRANEK, F. & SYKES, E. R. (2004): A prototype for an Intelligent Tutoring System for students learning to program in Java. In: Advanced Technology For Learning 1. S. 35-43. http://www.cas.mcmaster.ca/~franya/journals/wbe_extended_version.pdf. (Verifizierungsdatum 18.10.2005)

FRANK, E. & WITTEN, I.H. (2001): Data Mining. Praktische Werkzeuge und Techniken für das maschinelle Lernen. Aus dem Amerikanischen übersetzt. (2000, Practical Machine Learning Tools and Techniques with Java Implementation. San Fransisco: Morgan Kaufmann Publishers) München und Wien: Carl Hanser Verlag.

GILMORE, D. J., GREEN, T.R.G., HOC, J.-M., SAMURÇAY, R. (Hrsg.) (1990): Psychology of Programming. London et al.: Academic Press.

GO, S., IKEDA, M. & MIZOGUCHI, R. (1997): Opportunistic Group Formation. A Theory for Intelligent Support in Collaborative Learning. In: Du Boulay, B. & Mizoguchi, R. (1997) (Hrsg.): AI-ED'97 8th World Conference on Artifical Intelligence in Education. Amsterdam.

GÖLNDER, A. (2005): Computerunterstützte Kommunikation in virtuellen Teams. Klassifikations- und Lösungsansätze für Problemsituationen in der Chatkommunikation im Rahmen objektorientierter Programmierung. Magisterarbeit. Universität Hildesheim.

GÖßNER, J., STEIMANN, F. & TÖNNIES, S. (o. J. a): Projectory - ein Tool zur Unterstützung des Einsatzes von XP Techniken im universitären Programmierpraktikum. http://www.projectory.org/publications/ProjectoryXP.pdf. (Verifizierungsdatum 18.10.2005)

GÖßNER, J., STEIMANN, F. & MÜCK, T. (o. J. b): Projectory - A Tool for teaching Object Oriented Design and Object Oriented Programming. http://www.projectory.org/publications/Projectory.pdf. (Verifizierungsdatum 18.10.2005)

GRABOWSKI, B. & PENNINGTON, N. (1990): The Tasks of Programming. In: Gilmore et al. (1990). S. 45-62.

GRAMS, T. (1990): Denkfallen und Programmierfehler. Berlin et al.: Springer-Verlag.

GREEN, T.R.G. (1990a): The Nature of Programming. In: Gilmore et al. (1990), S. 21-44.

GREEN, T.R.G. (1990b): Programming Languages as Information Structures. In: Gilmore et al. (1990), S. 117-137.

GRIFFITHS, R., HOLLAND, S. & WOODMAN, M. (1997): Avoiding Object Misconceptions. SIGCSE'97. ACM, S. 131-134.

VAN HAASTER, K. & HAGAN, D. (2004): Teaching and Learning with BlueJ: An Evaluation of a Pedagogical Tool. In: Issues in Information Science and Information Technology. S. 455-470. www.bluej.org/papers/vanhaaster-hagan.pdf. (Verifizierungsdatum 18.10.2005)

HALLAND, K. & MALAN, K. (2004): Examples that can do Harm in Learning Programming. In: Proceedings OOPSLA'04. 24.–28. Oktober, 2004, Vancouver, Canada, ACM. S. 83-87.

HANCOCK, C., HOBBS, R., MARTIN, F., PERKINS, D.N., SIMMONS, R. (1989): Conditions of Learning in Novice Programmers. In: Soloway & Spohrer (1989), S. 261 – 279.

HAGAN, D., & MARKHAM, S. (2000). Teaching Java with the BlueJ Environment. Paper presented at the Australian Society for Computers in Learning in Tertiary Education (ASCILITE 2000), Coffs Harbour, Australia.

HERRMANN, J. (1997): Maschinelles Lernen und Wissensbasierte Systeme. Systematische Einführung mit praxisorientierten Fallstudien. Berlin et al.: Springer-Verlag.

HOC, J.-M. & NGUYEN-XUAN, A. (1990): Language Semantics, Mental Models and Analogy. In: Gilmore et al. (1990), S. 139-156.

HOLMBOE, C. (2005): The Linguistics of Object-Oriented Design: Implications for Teaching. In: Proceedings of the ITiCSE'05. 27.-29. Juni, 2005, Monte de Caparica, Portugal, ACM, S. 188-192.

HOVEMEYER, D. & PUGH, D. (2004): Finding Bugs is Easy. In: SIGPLAN Notices. 39, (12), OOPSLA onward!, S. 92-106.

HOVEMEYER, D., PUGH, D. & SPACCO, J. (2005): Understanding Bugs in Student Programming Projects. www.cs.umd.edu/~jspacco/marmoset/papers/fse2005.pdf. (Verifizierungsdatum 18.10.2005)

HRISTOVA, M., MISRA, A., MERCURI, R. & RUTTER, M. (2003): Identifying and Correcting Java Programming Errors for Introductory Computer Science Students**.** In: Proceedings of SIGCSE '03. 19.-23. Februar, 2003, Reno, Nevada, USA, S.153-156.

ISVL: http://kmi.open.ac.uk/people/paulm/isvl.html. (Verifizierungsdatum 18.10.2005)

IYENGAR, S. & SOLOWAY, E. (Hrsg.) (1986): Empirical Studies of Programmers. 1st Workshop, June 5-6, 1986 Washington D.C. Norwood NJ: Ablex Publishing Corporation.

JADUD, M. C. (2005): A First Look at Novice Compilation Behaviour using BlueJ. In: Computer Science Education. 15 (1), S. 25–40.

JERMANN, P., MUEHLENBROCK, M. & SOLLER, A. (2001): From Mirroring to Guiding: A Review of State of the Art Technology for Supporting Collaborative Learning. In: Dillenbourg, A. E. & Hakkarainen, K. (Hrsg.): European perspectives on computer-supported collaborative learning: Proceedings of the first European conference on computer-supported collaborative learning EuroCSCL - 2001 Maastricht: University of Maastricht, S. 324-331.

JLint: https://sourceforge.net/projects/jlint/. (Verifizierungsdatum 18.10.2005)

JOACHIMS, T. (2002): Learning to Classify Text using Support Vector Machines. Boston et al.: Kluwer Academic Publishers.

JOHNSON, W.L. (1986): Intention-Based Diagnosis of Novice Programming Errors. London et al.: Pitman et al. (Los Altos: Morgan Kaufmann Publishers).

JOHNSON, W.L. (1990): Understanding and Debugging Novice Programs. In: Clancey, W. J. & Soloway, E. (1990): Artificial Intelligence and Learning Environments. Cambridge & London: MIT Press, Amsterdam: Elsevier Science Publishers, S. 51-97.

JOHNSON, W.L. & SOLOWAY, E. (1984): PROUST: Knowledge-Based Program Understanding. In: Proceedings ICSE. 7th International Conference on Software Engineering, 26.-29. März, 1984, Orlando, Florida. IEEE Computer Society, S. 369-380.

JONES, R. M. & VANLEHN, K. (1993): What mediates the self-explanation effect? Knowledge gaps, schemas or analogies? In: Polson, M. (Hrsg.): Proceedings of the Fifteenth Annual Conference of the Cognitive Science Society. Hillsdale, NJ: Erlbaum, S. 1034- 1039.

JOP-PRAKTOMAT**:** http://jop.informatik.uni-wuerzburg.de/. (Verifizierungsdatum 18.10.2005)

JUNIT: www.junit.org. (Verifizierungsdatum 18.10.2005)

JÜRJENS, J., KOLLER, C., WAGNER, S. & TRISCHBERGER, P. (2005): Comparing Bug Finding Tools with Reviews and Tests. In. Khendek, F. & Dssouli, R. (Hrsg.): Testing of Communicating Systems, 17th IFIP TC6/WG 6.1 International Conference, TestCom 2005, Montreal, Canada, 31. Mai – 2. Juni, 2005, Proceedings. LNCS 3502 Springer 2005. S. 40-55 http://www4.informatik.tu-muenchen.de/publ/papers/SWJJCKPT05.pdf. (Verifizierungsdatum 18.10.2005)

KO, A. J. & MYERS, B. A. (2003): Development and Evaluation of a Model of Programming errors. In: Proceedings of the IEEE 2003 Symposia on Human Centric Computing Languages and Environments (HCC'03), Auckland, New Zealand, Oktober, 2003, S. 7-14.

KO, A. J. & MYERS, B. A. (2005): A framework and methodology for studying the causes of software errors in programming systems. In: Journal of Visual Languages and Computing. 16, S. 41–84.

KÖLLE, R. & LANGEMEIER, G. (2004a): Analyse und Unterstützung virtueller Lernteams bei der objektorientierten Softwareentwicklung. In: Bekavac, B., Herget, J. & Rittberger, M. (Hrsg.): Information zwischen Kultur und Marktwissenschaft. Proceedings des 9. Internationalen Symposiums für Informationswissenschaft (ISI 2004). Konstanz: UVK. (Schriften zur Informationswissenschaft; Bd. 42), S. 1-22.

KÖLLE, R. & LANGEMEIER, G. (2004b): Rollenorientierte Aspekte verteilter, synchroner Kollaboration bei der objektorientierten Softwareentwicklung. In: Rebensburg,

K.: Grundfragen Multimedialen Lehrens und Lernens. Proceedings des 2. Workshops GML 2004, S. 127-138.

KÖLLING, M., PATTERSON, A., QUIG, B. & ROSENBERG, J. (2003): The BlueJ System and its Pedagogy. In: Computer Science Education. Vol. 13, No 4 (2003), S. 249-268.

KRINKE, J., STÖRZER, M. & ZELLER, A.(2002): Web-basierte Programmierpraktika mit Praktomat. In: Softwaretechnik-Trends 22 (3), S. 51-53.

KRINKE, J. & ZELLER, A. (2004[2]): Open-Source-Programmierwerkzeuge. Versionskontrolle – Konstruktion – Testen – Fehlersuche. Heidelberg: dpunkt.verlag GmbH.

KRINKE, J., SNELTING, G. & ZELLER, A. (2001): Praktomat. Die Qualitätskontrolle im Programmierpraktikum. http://www.vhb.org/dokumente/Workshops/Workshop2001/Praktomat.pdf. (Verifizierungsdatum 18.10.2005)

LANE, H.C. & VANLEHN, K. (2004): A Dialogue-Based Tutoring System for Beginning Programming. In: Barr, V. & Markov, Z. (Hrsg.) (2004): Proceedings of the 17th International Florida Artificial Intelligence Research Society Conference, 17.-19. Mai, 2004, Miami Beach, Florida, USA. AAAI Press, S. 449-554.

LAVY, I. & OR-BACH, R (2004): Cognitive Activities of Abstraction in Object Orientation: An Empirical Study. In: Inroads – The SIGCSE Bulletin. 36 (2), S. 82-86.

LEWIS, S. F. & WATKINS, M. (2001): Using Java tools to teach Java, the integration of BlueJ and CourseMaster for delivery over the Internet. In: Proceedings of the 5th Java in the Computer Curriculum Conference (JCCC 5). South Bank University, UK.

LIEBERMAN, H. (HRSG.) (2001): Your Wish is my command. Programming by example. San Francisco: Morgan Kaufmann Publishers. Publisher Info: http://www.loc.gov/catdir/description/els031/00069638.html. (Verifizierungsdatum 18.10.2005)

LINN, M.C. & DALBEY, J. (1989). Cognitive consequences of programming instruction. In: Soloway & Spohrer (1989), S. 57–81.

MARTIN, F. & PERKINS, D.N. (1986): Fragile Knowledge and Neglected Strategies in Novice Programmers. In: Iyengar & Soloway (1986), S. 213-229.

MAYER, R.E. (HRSG.) (1988): Teaching and Learning Computer Programming. Multiple Research Perspectives. Hillsdale, NJ: Lawrence Erlbaum Associates.

MAYRBERGER, K. (2004): Kooperatives Lernen in der computerunterstützten Präsenzlehre der Hochschule. In: Krause, D., Oberquelle, H. & Pape, B. (Hrsg.) (2004): Wissensprojekte. Gemeinschaftliches Lernen aus didaktischer, softwaretechnischer und organisatorischer Sicht. Münster: Waxmann Verlag GmbH, S. 35-54.

MCMANUS, M. M. & AIKEN, R. M. (1995): Monitoring computer-based problem solving. In: Journal of Artificial Intelligence in Education. 6 (4), S. 307-336.

MICHIELS, I. & BÖRSTLER, J. (2000). ECOOP 2000 Workshop Report: Tools and Environments for Understanding Object-Oriented Concepts. Online: http://users.rcn.com/jcoplien/Ecoop2000Workshop.ps

MITCHELL, T. M. (1997): Machine Learning. Boston et al.: McGraw-Hill.

MITCHELL, T. M., KELLER, R.M. & KEDAR-CABELLI, S.T. (1986). Explanation-based generalization: A unifying view. In: Machine Learning, I, S. 47-80.

MÖBUS, C. & SCHRÖDER, O. (1994): Wissensdiagnostik und Extraktion von Handlungsstrategien aus Verbaldaten auf der Basis der ISPDL-Theorie. http://lls.informatik.uni-oldenburg.de/dokumente/1994/94_bibb.ps. (Verifizierungsdatum 18.10.2005)

MYERS, B.A. (1986): Visual Programming, Programming by Example, and Program Visualization: A Taxonomy. In: Proceedings CHI'86. April 1986, S. 59-66.

NEAL, L.R. (1989): A System for Example-Based Programming. In: Proceedings CHI'89. ACM, S. 63-68.

NUMAO, M., SHIMURA, M. & SISON, R.C. (2000): Multistrategy Discovery and Detection of Novice Programmer Errors. In: Machine Learning, 38, 2000, Kluwer Academic Publishers: Niederlande S. 157-180.

ORMEROD, T. (1990): Human Cognition and Programming. In: Gilmore et al. (1990), S. 63-82.

PENNINGTON, N. (1987): Comprehension strategies in programming. In: G. M. Olson, S. Sheppard, and E. Soloway (Hrsg.), Empirical Studies of Programmers: Second Workshop. Norwood, NJ: Ablex, S. 100-113.

PERKINS, D. N., SIMMONS, R., & SCHWARTZ, S. (1988): Instructional Strategies for the Problems of Novice Programmers. In: Mayer (1988), S. 153-178.

PETRE, M. (1990): Expert Programmers and Programming Languages. In: Gilmore et al. (1990), S. 103-115.

PFISTER, H.-R., & WESSNER, M. (2001): Kooperatives Lehren und Lernen. In: Schwabe, G., Streitz, R. & Unland, R. (2001): CSCW-Kompendium. Lehr-und Handbuch zum computerunterstützten kooperativen Arbeiten. Berlin et al.: Springer, S. 251-263.

PILLAY, N. (2003): Developing Intelligent Programming Tutors for Novice Programmers. In: inroads –The SIGCSE Bulletin. 35 (2), S. 78-82.

PMD: http://pmd.sourceforge.net/ (Verifizierungsdatum 18.10.2005)

POPE, E., SOLOWAY, E. & SPOHRER, J.C. (1989): A Goal/Plan Analysis of Buggy Pascal Programs. In: Soloway & Spohrer (1989), S. 355-399.

PUTNAM, R. T., SLEEMAN, D., BAXTER, J. A. & KUSPA, L. K. (1989): A Summary of Misconceptions of High School BASIC Programmers. In: Soloway & Spohrer (1989), S. 301-314.

QUI, L. (2004): Intelligent Educational Systems for Teaching Programming. http://www.cs.oswego.edu/~lqiu/critiquer/publications/acm2004.pdf (Verifizierungsdatum 18.10.2005).

RAMADHAN, H. A. (1992): An Intelligent Discovery Programming System. In: Proceedings of the 1992 ACM/SIGAPP Symposium on Applied computing: technological challenges of the 1990's. S. 149-159.

RAMADHAN, H. A. & SHIHAB, K. (2000): Intelligent Systems for Active Program Diagnosis. In: Science and Education, Special Review (2000), S. 157-183.

RAMALINGAM, V. & WIEDENBECK, S. (1997): An Empirical Study of Novice Program Comprehension in the Imperative and Object-Oriented Styles. In: Empirical

Studies of Programmers - Seventh Workshop. 24.-26. Oktober, 1997, Alexandria, Virginia, S. 124-139.

REASON, J. (1994): Menschliches Versagen: psychologische Risikofaktoren und moderne Technologien. Übersetzung der amerikanischen Originalausgabe ‚Human Error'. Heidelberg et al.: Spektrum Akademischer Verlag

REIMANN, P. & SCHULT, T. J. (1996): Turning Examples into Cases: Acquiring Knowledge Structures for Analogical Problem Solving. In: Educational Psychologist, 31 (2), S. 123-132

RIST, R.S. (1986): Plans in Programming: Definition, Demonstration, and Development. In: Iyengar & Soloway (1986), S. 28-47

ROBINS, A., ROUNTREE, J., & ROUNTREE, N. (2003): Learning and Teaching Programming: A Review and Discussion. In: Computer Science Education. 13 (2), S. 137-172.

ROGALSKI, J. & SAMURÇAY, R. (1990): Acquisition of Programming Knowledge and Skills. In: Gilmore et al. (1990), S. 157-174.

ROHDE, MARTIN (2004): Studentische Praxisgemeinschaften in der Angewandten Informatik. In: Krause, D., Oberquelle, H. & Pape, B. (Hrsg.) (2004): Wissensprojekte. Gemeinschaftliches Lernen aus didaktischer, softwaretechnischer und organisatorischer Sicht. Münster: Waxmann Verlag GmbH, S. 90-107.

ROSCHELLE, J., DISESSA, A. A. & SMITH, J. P. (1993): Misconceptions Reconceived: A Constructivist Analysis of Knowledge in Transition. In: Journal of the Learning Sciences, 1993-1994, Vol. 3, No. 2, S. 115-163.

SAMURÇAY, R. (1989): The Concept of Variable in Programming: Its Meaning and Use in Problem Solving by Novice Programmers. In: Soloway & Spohrer (1989), S. 161-178.

SCHIEDERMEIER, R. (1996): Vorlesung "Programmieren I". Vorlesungsunterlagen. FH München, FB Informatik/Mathematik. http://www.informatik.fh-muenchen.de/~schieder/programmieren-1-ws96-97/sprachen.html. (Verifizierungsdatum 18.10.2005)

SCHOOLER, L. J. & ANDERSON, J. R. (1990): The disruptive potential of immediate feedback. In: Proceedings of the Twelfth Annual Conference of the Cognitive Science Society 1990, S. 702 - 708.

SCHULMEISTER, R. (1997[2]): Grundlagen hypermedialer Lernsysteme. 2., aktualisierte Auflage. München und Wien: Oldenbourg Verlag.

SHEIL, B. A. (1981): The Psychological Study of Programming. In: Computing Surveys, Vol. 13, No.1 (März 1981), S. 101–120.

SHIMURA, M. & SISON, R. (1998): Student Modeling and Machine Learning. In: International Journal of Artifical Intelligence in Eduacation. 9, S. 128-158.

SHNEIDERMAN, B. (1986): Empirical studies of Programmers: The Territory, Paths, and Destinations. In: Iyengar & Soloway (1986), S. 1-12.

SIEBERT, HORST (2003[2]): Pädagogischer Konstruktivismus. Lernen als Konstruktion von Wirklichkeit. 2., vollständig überarbeitete und erweiterte Auflage. München/ Unterschleißheim: Luchterhand.

SIMON, H. (1983): Why should machines learn? In: Michalski, R. S., Carbonell, J. G. & Mitchell, T. M.: Machine Learning: An Artificial Intelligence Approach. Berlin: Springer, S. 25-37.

SLAVIN, R. (1990): Cooperative Learning: Theory, research and practice. Prentice Hall: NJ.

SOLOWAY, E., SPOHRER, J. C. & POPE, E. (1985): Where The Bugs Are. In: Proceedings CHI'85. April 1985, ACM, S. 47-53.

SOLOWAY, E. & SPOHRER, J. C. (1986a): Analyzing the High Frequency Bugs in Novice Programs. In: Iyengar & Soloway (1986), S. 230-250.

SOLOWAY, E. & SPOHRER, J. C. (1986b): Novice Mistakes: Are the folks wisdom correct? In: Communications of the ACM. 29 (7), S. 624-632.

SOLOWAY, E. & SPOHRER, J. C. (Hrsg.) (1989): Studying the Novice Programmer. Hillsdale, NJ: Lawrence Erlbaum Associates.

THEFREECOUNTRY**:** Free Java Editors and IDE. Free Java Integrated Development Environment. http://www.thefreecountry.com/programming/javaide.shtml. (Verifizierungsdatum 18.10.2005)**.**

UNIVERSITY OF STIRLING: University of Stirling: Using RealJ. Basic Demo. http://www.cs.stir.ac.uk/~sbj/tips/Javainfo/RealJ/Using-RealJ.pdf. (Verifizierungsdatum 18.10.2005)

VANLEHN, K. (1990): Mind Bugs. The Origins of Procedural Misconceptions. Cambridge: MIT Press.

WEBER, G. (1994): Fallbasiertes Lernen und Analogien. Unterstützung von Problemlöse- und Lernprozessen in einem adaptiven Lernsystem. Weinheim: Beltz, Psychologie Verlags Union.

WEBER, G. (1996): Episodic Learner Modeling. In: Cognitive Science. 20, S. 195-236.

WEINBERG, G. M. (2004): Die Psychologie des Programmierers. Seine Persönlichkeit, sein Team, sein Projekt. Jubiläumsausgabe, aus dem englischen Original von 1971. Bonn: mitp-Verlag.

WEYERHÄUSER, M. (2003): Die Programmierumgebung Eclipse. Teile und Herrsche. In: Javaspektrum. 02/2003, S. 12-17.

WILLIAMS, L., WIEBE, E., YANG, K., FERZLI, M. & MILLER, C. (2002): In Support of Pair Programming in the Introductory Computer Science Course. In: Computer Science Education. 12 (3), S. 197-212.

WINSLOW, L. E. (1996): Programming Pedagogy – A Psychological Overview. In: SIGSE Bulletin. 28 (3), ACM, S. 17-25.

WOMSER-HACKER, C. (2004): Einführung in die Informationswissenschaften. Vorlesungsskript. ZFW-Moodle. http://lms1.zfw.uni-hildesheim.de/moodle/course/view.php?id=34. (Verifizierungsdatum 18.10.2005)

ZIELHOFER, S. (2003): Aufgaben- und Feedbackgestaltung bei der Entwicklung eines multimedialen Lernsystems zum Thema Fakteninformationssysteme im Projekt SELIM. Magisterarbeit. Universität Hildesheim. Juni 2003.

ZIMBARDO, P. G.(1995): Psychologie. Berlin, Heidelberg: Springer Verlag.

ZUMBACH, J. (2003): Problembasiertes Lernen. Münster: Waxmann Verlag GmbH.

ABBILDUNGSVERZEICHNIS

EIGENSTÄNDIGKEITSERKLÄRUNG

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Außerdem versichere ich, dass die Arbeit noch nicht veröffentlicht oder in einem anderen Prüfungsverfahren als Prüfungsleistung vorgelegt wurde.

Hildesheim, im Oktober 2005

_____